DEC Ada

Language Reference Manual

Order Number: AA–PYZAB–TK

June 1995

This manual represents the Digital-supplemented text of ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language*. Textual insertions describe the Digital interpretation of implementation-dependent language features, as well as allowed implementation-specific additions to the language (pragmas, attributes, input-output features, and so on).

If you have any comments on this or any other DEC Ada document, please send them to the following Internet address:

ada_docs@zko.mts.dec.com

Digital Equipment Corporation
Maynard, Massachusetts

Contents

3 Declarations and Types

4 Names and Expressions

5 Statements

6 Subprograms

7 Packages

8 Visibility Rules

9 Tasks

10 Program Structure and Compilation Issues

11 Exceptions

12 Generic Units

13 Representation Clauses and Implementation-Dependent Features

14 Input-Output

A Predefined Language Attributes

B Predefined Language Pragmas

C Predefined Language Environment

D Glossary

E Syntax Summary

F Implementation-Dependent Characteristics

G Ada Language Interpretations

Index

Postscript : Submission of Comments

Figures

Preface

The entire text of the *Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A-1983, ISO/8652-1987) is reprinted in this manual. In addition, this manual contains DEC Ada implementation information and Digital-supplied supplementary examples and text. The Digital-supplied information covers DEC Ada implementations across a number of systems. Information common to all systems is identified as DEC Ada information. System-specific information is identified with the name of the system to which it applies.

DEC Ada information appears in chapters 1, 2, 3, 4, 6, 9, 10, 11, 12, 13, and 14, Annexes A, B, and C, and Appendices D and F. Appendix G, added by Digital, lists further interpretations of the standard Ada language that have been made between the publication of the standard and the publication of this manual. Footnotes referring to that appendix appear throughout this manual.

Intended Audience

This manual is intended for all programmers who are designing or implementing applications using Ada. Readers should understand the concepts of programming in Ada and should have some familiarity with the system on which their Ada compiler is running.

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

Operating System

For more information on your operating system →

Operating System Documentation

Other Layered Products

For more information on various layered products →

Individual Layered Product Documentation

Ada Language

For introductory or tutorial Ada language information →

Commercial Ada Textbooks
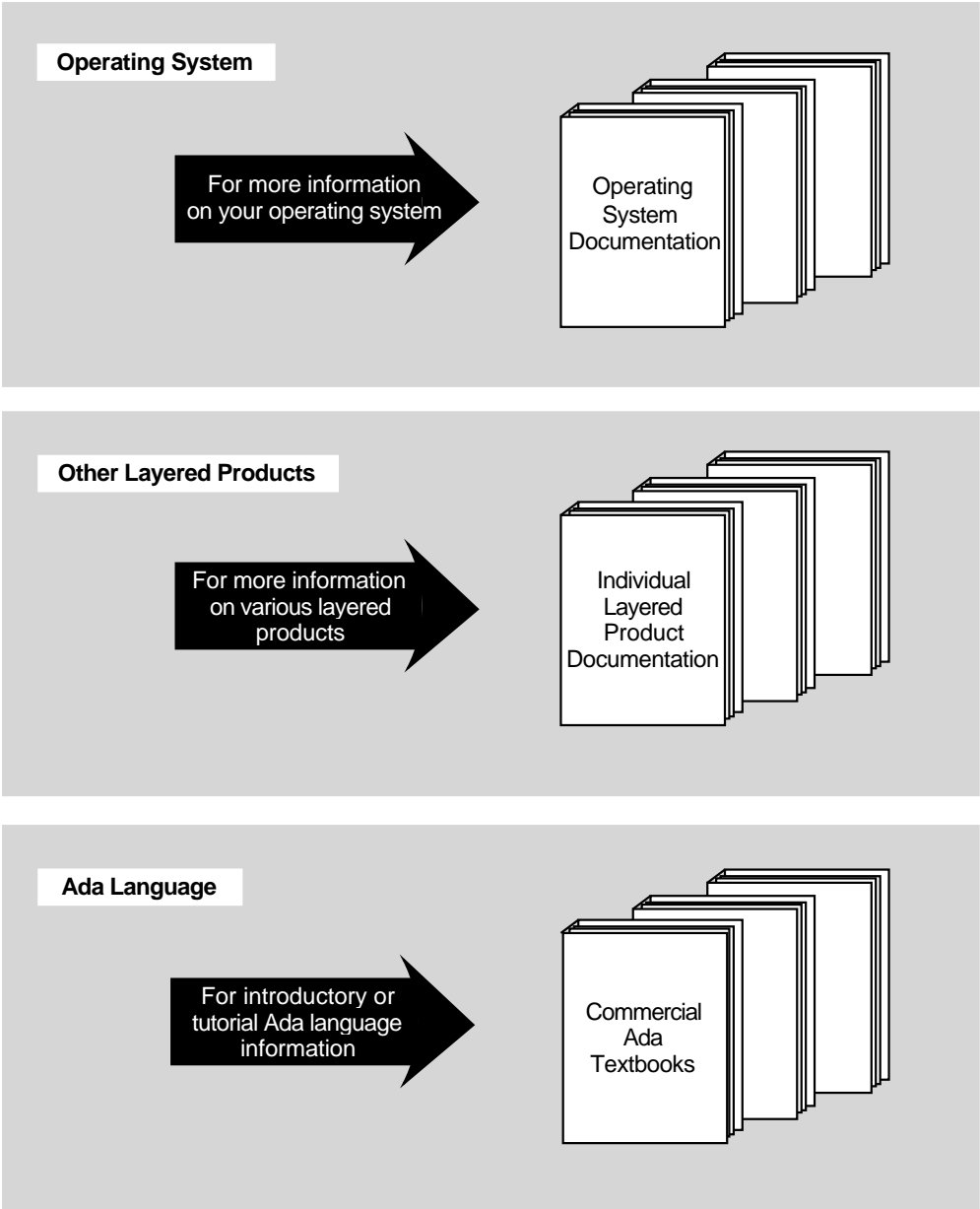
ZK−6003A−GE

Figure 2 Documentation Reading Path for DEC Ada Documentation

DEC Ada

Developing and compiling →

DEC Ada
Developing Ada Programs
*

Implementation and run–time details →

DEC Ada
Run–Time Reference Manual
*

Designing and imple–menting applications →

DEC Ada
Language Reference Manual

Installing →

DEC Ada
Installation Guide
*

* Operating System–Specific Manual
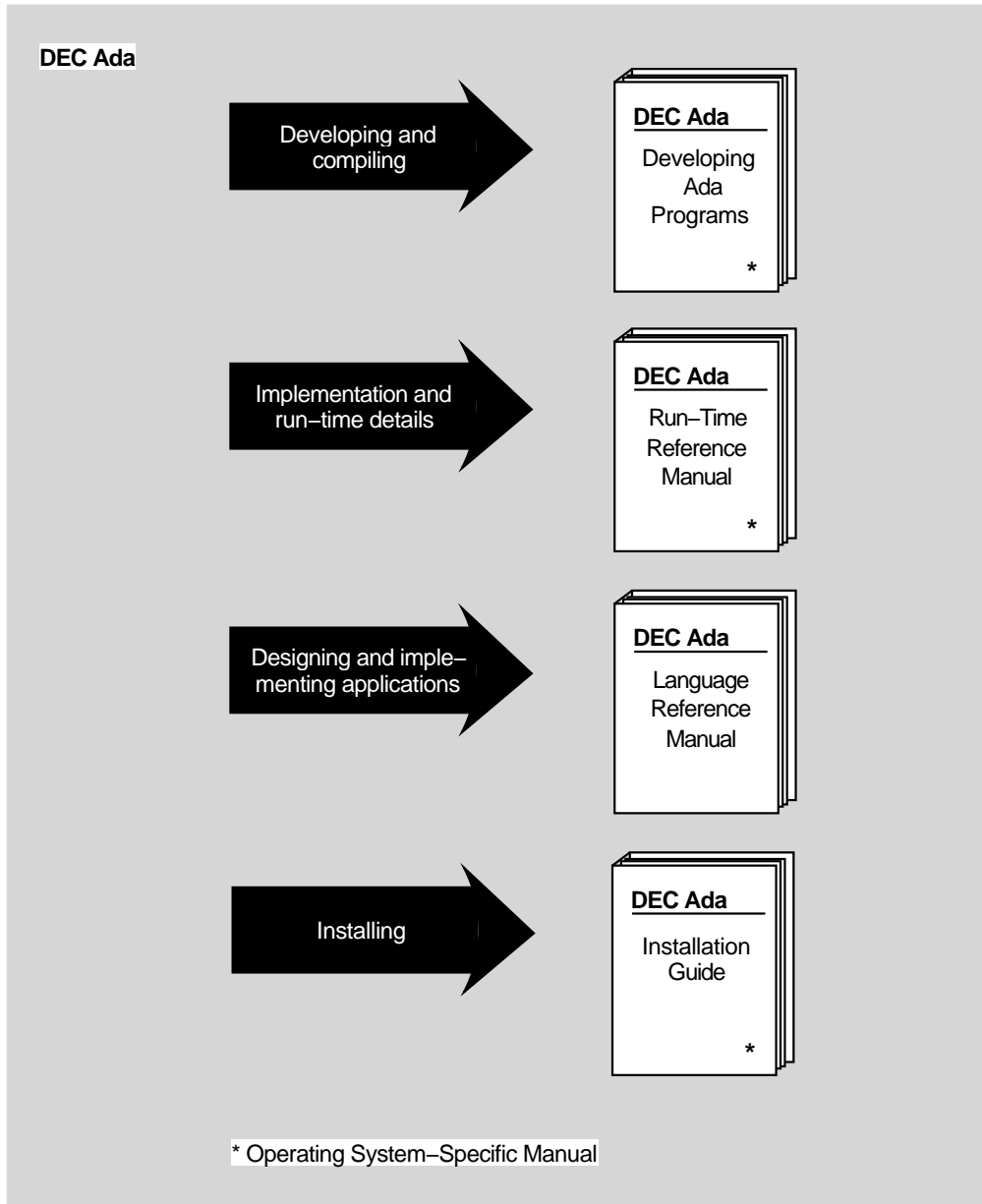
ZK–5349A–1–GE

Document Structure

This manual has the following structure:

- Chapter 1 contains a description of the Ada language standard, a language overview, a characterization of DEC Ada, and a description of the syntax notation.

- Chapter 2 provides detailed information on the lexical elements, and notes the DEC Ada implementation of the Latin-1 character set.

- Chapter 3 describes Ada types and the rules for declaring constants, variables, and named numbers. It gives the additional DEC Ada integer and floating point types, and describes the DEC Ada pragmas FLOAT_REPRESENTATION and LONG_FLOAT.

- Chapter 4 gives the rules for names and expressions.

- Chapter 5 gives the general rules that apply to all Ada statements, as well as the syntax and semantics of most of those statements.

- Chapter 6 gives the rules relating to subprograms, and notes the DEC Ada implementation of the pragma INLINE.

- Chapter 7 gives the rules relating to packages.

- Chapter 8 gives the rules defining the scope of declarations, as well as the rules defining the visibility of identifiers at various points in the text of a program.

- Chapter 9 explains Ada tasks. It also notes the DEC Ada implementation of the pragma SHARED and describes the DEC Ada pragmas TIME_SLICE and VOLATILE, as well as the DEC Ada pragma and attribute AST_ENTRY.

- Chapter 10 explains the overall structure of programs and the facilities for separate compilation.

- Chapter 11 defines the facilities for dealing with errors or exceptions that arise during program execution. It notes the DEC Ada treatment of the pragma SUPPRESS and presents the DEC Ada pragma SUPPRESS_ALL.

- Chapter 12 explains the use of generic units and the process of instantiation. It also presents the DEC Ada pragmas INLINE_GENERIC and SHARE_GENERIC.

- Chapter 13 describes representation clauses and certain DEC Ada features for systems programming. It describes the DEC Ada interpretations of the pragma PACK, the attributes SIZE, STORAGE_SIZE, and SMALL, and presents the DEC Ada pragmas COMPONENT_ALIGNMENT, MAIN_STORAGE, and TASK_STORAGE.

  Chapter 13 also gives the DEC Ada additions to the package SYSTEM, the DEC Ada interpretations of the representation attributes ADDRESS and SIZE, and describes the DEC Ada representation attributes BIT and MACHINE_SIZE.

  Finally, chapter 13 presents the DEC Ada interpretation of the pragma INTERFACE and describes the DEC Ada pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, IMPORT_VALUED_PROCEDURE, INTERFACE_NAME, EXPORT_FUNCTION, EXPORT_PROCEDURE, EXPORT_VALUED_PROCEDURE, COMMON_OBJECT, IMPORT_OBJECT, EXPORT_OBJECT, PSECT_OBJECT, IMPORT_EXCEPTION, and EXPORT_EXCEPTION.

- Chapter 14 gives detailed information on the standard input-output packages (SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO). It also presents the DEC Ada input-output packages (RELATIVE_IO, INDEXED_IO, SEQUENTIAL_MIXED_IO, RELATIVE_MIXED_IO, INDEXED_MIXED_IO, and DIRECT_MIXED_IO) and the DEC Ada package AUX_IO_EXCEPTIONS.

- Annex A summarizes the language attributes.

- Annex B summarizes all pragmas and defines the standard pragmas LIST, OPTIMIZE, and PAGE. It also defines the DEC Ada pragmas IDENT and TITLE.

- Annex C presents the specification of the package STANDARD.

- Appendix D is a glossary of Ada terms. It is not part of the standard definition of the Ada language.

- Appendix E contains a syntax summary of the Ada language. It is not part of the standard definition of the Ada language.

- Appendix F lists the DEC Ada implementation-dependent characteristics. It is not part of the standard definition of the Ada language.

- Appendix G presents summaries of Ada language interpretations made or recommended between the publication of the *Reference Manual for the Ada Programming Language* and the publication of this edition of the *DEC Ada Language Reference Manual*. It is not part of the standard definition of the Ada language.

Conventions

This manual uses the conventions described in section 1.5. The Ada language syntax is described using a simple variant of Backus-Naur-Form.

Colored print distinguishes DEC Ada insertions in hardcopy versions of this manual; shading distinguishes DEC Ada insertions in online versions.

DEC Ada is available on a number of systems. The following abbreviated terms are used to refer to those systems throughout this manual.

System abbreviation	Specific systems it refers to
OpenVMS	VMS OpenVMS VAX OpenVMS Alpha
Alpha	OpenVMS Alpha Digital UNIX® Alpha
Digital UNIX	Digital UNIX Alpha

1

Introduction

1 Ada is a programming language designed in accordance with requirements defined by the United States Department of Defense: the so-called Steelman requirements. Overall, these requirements call for a language with considerable expressive power covering a wide application domain. As a result, the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages. Thus the language is a modern algorithmic language with the usual control structures, and with the ability to define types and subprograms. It also serves the need for modularity, whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation.

2 In addition to these aspects, the language covers real-time programming, with facilities to model parallel tasks and to handle exceptions. It also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, both application-level and machine-level input-output are defined.

1.1 Scope of the Standard

1 This standard specifies the form and meaning of program units written in Ada. Its purpose is to promote the portability of Ada programs to a variety of data processing systems.

1.1.1 Extent of the Standard

1 This standard specifies:

2   (a) The form of a program unit written in Ada.

3   (b) The effect of translating and executing such a program unit.

4   (c) The manner in which program units may be combined to form Ada programs.

5   (d) The predefined program units that a conforming implementation must supply.

6   (e) The permissible variations within the standard, and the manner in which they must be specified.

7   (f) Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program unit containing such violations.

8   (g) Those violations of the standard that a conforming implementation is not required to detect.

9   This standard does not specify:

10   (h) The means whereby a program unit written in Ada is transformed into object code executable by a processor.

11   (i) The means whereby translation or execution of program units is invoked and the executing units are controlled.

12   (j) The size or speed of the object code, or the relative execution speed of different language constructs.

13   (k) The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages.

14   (l) The effect of executing a program unit that contains any violation that a conforming implementation is not required to detect.

15   (m) The size of a program or program unit that will exceed the capacity of a particular conforming implementation.

16   Where this standard specifies that a program unit written in Ada has an exact effect, this effect is the operational meaning of the program unit and must be produced by all conforming implementations. Where this standard specifies permissible variations in the effects of constituents of a program unit written in Ada, the operational meaning of the program unit as a whole is understood to be the range of possible effects that result from all these variations, and a conforming implementation is allowed to produce any of these possible effects. Examples of permissible variations are:

17   • The represented values of fixed or floating numeric quantities, and the results of operations upon them.

18   • The order of execution of statements in different parallel tasks, in the absence of explicit synchronization.

1.1.2 Conformity of an Implementation With the Standard

1 A conforming implementation is one that:[1]

2 (a) Correctly translates and executes legal program units written in Ada, provided that they are not so large as to exceed the capacity of the implementation.

3 (b) Rejects all program units that are so large as to exceed the capacity of the implementation.

4 (c) Rejects all program units that contain errors whose detection is required by the standard.

5 (d) Supplies all predefined program units required by the standard.

6 (e) Contains no variations except where the standard permits.

7 (f) Specifies all such permitted variations in the manner prescribed by the standard.

1.2 Structure of the Standard

1 This reference manual contains fourteen chapters, three annexes, three appendices, and an index.

 This manual adds a fourth appendix, Appendix G, which summarizes Ada interpretations made or recommended between the publication of the *Reference Manual for the Ada Programming Language* and the publication of this edition of the *DEC Ada Language Reference Manual.*

2 Each chapter is divided into sections that have a common structure. Each section introduces its subject, gives any necessary syntax rules, and describes the semantics of the corresponding language constructs. Examples and notes, and then references, may appear at the end of a section.

3 Examples are meant to illustrate the possible forms of the constructs described. Notes are meant to emphasize consequences of the rules described in the section or elsewhere. References are meant to attract the attention of readers to a term or phrase having a technical meaning defined in another section.

4 The standard definition of the Ada programming language consists of the fourteen chapters and the three annexes, subject to the following restriction: the material in each of the items listed below is informative, and not part of the standard definition of the Ada programming language:

5 • Section 1.3 Design goals and sources

---

[1] See also Appendix G, AI-00325.

6   • Section 1.4 Language summary

7   • The examples, notes, and references given at the end of each section

8   • Each section whose title starts with the word "Example" or "Examples"

1.3 Design Goals and Sources

1   Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency.

2   The need for languages that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared and that their type be specified. Since the type of a variable is invariant, compilers can ensure that operations on variables are compatible with the properties intended for objects of the type. Furthermore, error-prone notations have been avoided, and the syntax of the language avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and maintenance, and which provides the same degree of checking between units as within a unit.

3   Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep the language as small as possible, given the ambitious nature of the application domain. We have attempted to cover this domain with a small number of underlying concepts integrated in a consistent and systematic way. Nevertheless we have tried to avoid the pitfalls of excessive involution, and in the constant search for simpler designs we have tried to provide language constructs that correspond intuitively to what the users will normally expect.

4   Like many other human activities, the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components has been a central idea in this design. The concepts of packages, of private types, and of generic units are directly related to this idea, which has ramifications in many other aspects of the language.

5   No language can avoid the problem of efficiency. Languages that require over-elaborate compilers, or that lead to the inefficient use of storage or execution time, force these inefficiencies on all machines and on all programs. Every construct of the language was examined in the light of present implementation techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected.

6   None of the above design goals was considered as achievable after the fact. The design goals drove the entire design process from the beginning.

7   A perpetual difficulty in language design is that one must both identify the capabilities required by the application domain and design language features that provide these capabilities. The difficulty existed in this design, although to a lesser degree than usual because of the Steelman requirements. These requirements often simplified the design process by allowing it to concentrate on the design of a given system providing a well defined set of capabilities, rather than on the definition of the capabilities themselves.

8   Another significant simplification of the design work resulted from earlier experience acquired by several successful Pascal derivatives developed with similar goals. These are the languages Euclid, Lis, Mesa, Modula, and Sue. Many of the key ideas and syntactic forms developed in these languages have counterparts in Ada. Several existing languages such as Algol 68 and Simula, and also recent research languages such as Alphard and Clu, influenced this language in several respects, although to a lesser degree than did the Pascal family.

9   Finally, the evaluation reports received on an earlier formulation (the Green language), and on alternative proposals (the Red, Blue, and Yellow languages), the language reviews that took place at different stages of this project, and the thousands of comments received from fifteen different countries during the preliminary stages of the Ada design and during the ANSI canvass, all had a significant impact on the standard definition of the language.

1.4 Language Summary

1   An Ada program is composed of one or more program units. These program units can be compiled separately. Program units may be subprograms (which define executable algorithms), package units (which define collections of entities), task units (which define parallel computations), or generic units (which define parameterized forms of packages and subprograms). Each unit normally consists of two parts: a specification, containing the information that must be visible to other units, and a body, containing the implementation details, which need not be visible to other units.

2   This distinction of the specification and body, and the ability to compile units separately, allows a program to be designed, written, and tested as a set of largely independent software components.

3 An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. The text of a separately compiled program unit must name the library units it requires.

4 **Program Units**

5 A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the means of invoking a series of actions. For example, it may read data, update variables, or produce some output. It may have parameters, to provide a controlled means of passing information between the procedure and the point of call.

6 A function is the means of invoking the computation of a value. It is similar to a procedure, but in addition will return a result.

7 A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a common pool of data and types, a collection of related subprograms, or a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification.

8 A task unit is the basic unit for defining a task whose sequence of actions may be executed in parallel with those of other tasks. Such tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit may define either a single executing task or a task type permitting the creation of any number of similar tasks.

9 **Declarations and Statements**

10 The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit.

11 The declarative part associates names with declared entities. For example, a name may denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, task units, and generic units to be used in the program unit.

12 The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession (unless an exit, return, or goto statement, or the raising of an exception, causes execution to continue from another place).

13 An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any actual parameters provided at the call with the corresponding formal parameters.

14   Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition.

15   The loop statement provides the basic iterative mechanism in the language. A loop statement specifies that a sequence of statements is to be executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered.

16   A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements.

17   Certain statements are only applicable to tasks. A delay statement delays the execution of a task for a specified duration. An entry call statement is written as a procedure call statement; it specifies that the task issuing the call is ready for a rendezvous with another task that has this entry. The called task is ready to accept the entry call when its execution reaches a corresponding accept statement, which specifies the actions then to be performed. After completion of the rendezvous, both the calling task and the task having the entry may continue their execution in parallel. One form of the select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls.

18   Execution of a program unit may encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may be made to access an array component by using an incorrect index value. To deal with such error situations, the statements of a program unit can be textually followed by exception handlers that specify the actions to be taken when the error situation arises. Exceptions can be raised explicitly by a raise statement.

19   **Data Types**

20   Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main classes of types are scalar types (comprising enumeration and numeric types), composite types, access types, and private types.

21   An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types BOOLEAN and CHARACTER are predefined.

22   Numeric types provide a means of performing exact or approximate numerical computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bounds on the error, or floating point types, with relative bounds

on the error. The numeric types INTEGER, FLOAT, and DURATION are predefined.

23 Composite types allow definitions of structured objects with related components. The composite types in the language provide for arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. The array type STRING is predefined.

24 A record may have special components called discriminants. Alternative record structures that depend on the values of discriminants can be defined within a record type.

25 Access types allow the construction of linked data structures created by the evaluation of allocators. They allow several variables of an access type to designate the same object, and components of one object to designate the same or other objects. Both the elements in such a linked data structure and their relation to other elements can be altered during program execution.

26 Private types can be defined in a package that conceals structural details that are externally irrelevant. Only the logically necessary properties (including any discriminants) are made visible to the users of such types.

27 The concept of a type is refined by the concept of a subtype, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values.

28 **Other Facilities**

29 Representation clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

30 Input-output is defined in the language by means of predefined library packages. Facilities are provided for input-output of values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

31 Finally, the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects) and so allow general algorithms to be applied to all types of a given class.

1.4a DEC Ada

All of the language elements specified by the *ANSI* or *ISO* standard definition for the Ada language are provided by DEC Ada. In addition, DEC Ada implements certain options and makes certain interpretations, as permitted by the standard. Material has been inserted throughout this manual to describe and explain these permitted options and interpretations. The term DEC Ada and colored print (hard copy) or shading (online) are used to distinguish the DEC Ada material.

1.5 Method of Description and Syntax Notation

1   The form of Ada program units is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules.

2   The meaning of Ada program units is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. This narrative employs technical terms whose precise definition is given in the text (references to the section containing the definition of a technical term appear at the end of each section that uses the term).

3   All other terms are in the English language and bear their natural meaning, as defined in Webster's Third New International Dictionary of the English Language.

4   The context-free syntax of the language is described using a simple variant of Backus-Naur-Form. In particular,

5   (a)   Lower case words, some containing embedded underlines, are used to denote syntactic categories, for example:

```
adding_operator
```

6   Whenever the name of a syntactic category is used apart from the syntax rules themselves, spaces take the place of the underlines (thus: adding operator).

7   (b)   Boldface words are used to denote reserved words, for example:

```
array
```

8   (c)   Square brackets enclose optional items. Thus the two following rules are equivalent.

```
                       return_statement ::= return [expression];
                       return_statement ::= return; | return expression;
```

9   (d)   Braces enclose a repeated item. The item may appear zero or more
             times; the repetitions occur from left to right as with an equivalent
             left-recursive rule. Thus the two following rules are equivalent.

```
                  term ::= factor {multiplying_operator factor}
                  term ::= factor | term multiplying_operator factor
```

10   (e)   A vertical bar separates alternative items unless it occurs immediately
             after an opening brace, in which case it stands for itself:

```
                  letter_or_digit ::= letter | digit
                  component_association ::=
                     [choice {| choice} =>] expression
```

11   (f)   If the name of any syntactic category starts with an italicized part,
             it is equivalent to the category name without the italicized part.
             The italicized part is intended to convey some semantic information.
             For example *type*_name and *task*_name are both equivalent to name
             alone.

Note:

12   The syntax rules describing structured constructs are presented in a form that
     corresponds to the recommended paragraphing. For example, an if statement
     is defined as

```
     if_statement ::=
         if condition then
           sequence_of_statements
         {elsif condition then
           sequence_of_statements}
         [else
           sequence_of_statements]
          end if;
```

13   Different lines are used for parts of a syntax rule if the corresponding
     parts of the construct described by the rule are intended to be on different
     lines. Indentation in the rule is a recommendation for indentation of the
     corresponding part of the construct. It is recommended that all indentations
     be by multiples of a basic step of indentation (the number of spaces for the
     basic step is not defined). The preferred places for other line breaks are after
     semicolons. On the other hand, if a complete construct can fit on one line, this
     is also allowed in the recommended paragraphing.

1.6 Classification of Errors

1   The language definition classifies errors into several different categories:

2   (a)   Errors that must be detected at compilation time by every Ada compiler.

3   These errors correspond to any violation of a rule given in this reference manual, other than the violations that correspond to (b) or (c) below. In particular, violation of any rule that uses the terms *must*, *allowed*, *legal*, or *illegal* belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, per se, that the program is free from other forms of error.

4   (b)   Errors that must be detected at run time by the execution of an Ada program.

5   The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. If an exception is certain to be raised in every execution of a program, then compilers are allowed (although not required) to report this fact at compilation time.

6   (c)   Erroneous execution.

7   The language rules specify certain rules to be obeyed by Ada programs, although there is no requirement on Ada compilers to provide either a compilation-time or a run-time detection of the violation of such rules. The errors of this category are indicated by the use of the word *erroneous* to qualify the execution of the corresponding constructs. The effect of erroneous execution is unpredictable.

8   (d)   Incorrect order dependences.

9   Whenever the reference manual specifies that different parts of a given construct are to be executed *in some order that is not defined by the language*, this means that the implementation is allowed to execute these parts in any given order, following the rules that result from that given order, but not in parallel. Furthermore, the construct is incorrect if execution of these parts in a different order would have a different effect. Compilers are not required to provide either compilation-time or run-time detection of incorrect order dependences. The foregoing is expressed in terms of the process that is called execution; it applies equally to the processes that are called evaluation and elaboration.

₁₀ If a compiler is able to recognize at compilation time that a construct is erroneous or contains an incorrect order dependence, then the compiler is allowed to generate, in place of the code otherwise generated for the construct, code that raises the predefined exception PROGRAM_ERROR. Similarly, compilers are allowed to generate code that checks at run time for erroneous constructs, for incorrect order dependences, or for both. The predefined exception PROGRAM_ERROR is raised if such a check fails.

2

Lexical Elements

1 The text of a program consists of the texts of one or more compilations. The text of a compilation is a sequence of lexical elements, each composed of characters; the rules of composition are given in this chapter. Pragmas, which provide certain information for the compiler, are also described in this chapter.

2 **References:** character 2.1, compilation 10.1, lexical element 2.2, pragma 2.8

2.1 Character Set

1 The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the *ISO* seven-bit coded character set (*ISO* standard 646), and is represented (visually) by a graphical symbol. Some graphic characters are represented by different graphical symbols in alternative national representations of the *ISO* character set. The description of the language definition in this standard reference manual uses the *ASCII* graphical symbols, the *ANSI* graphical representation of the *ISO* character set.[1]

2
```
graphic_character ::= basic_graphic_character
    | lower_case_letter | other_special_character

basic_graphic_character ::=
    upper_case_letter | digit
    | special_character | space_character

basic_character ::=
    basic_graphic_character | format_effector
```

3 The basic character set is sufficient for writing any program. The characters included in each of the categories of basic graphic characters are defined as follows:

4   (a) upper case letters A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

5   (b) digits 0 1 2 3 4 5 6 7 8 9

---

[1] See also Appendix G, AI-00339 and AI-00866.

6   (c)   special characters " # & ' ( ) * + , − . / : ; < => _ |

7   (d)   the space character

8   Format effectors are the *ISO* (and *ASCII*) characters called horizontal
    tabulation, vertical tabulation, carriage return, line feed, and form feed.

9   The characters included in each of the remaining categories of graphic
    characters are defined as follows:

10   (e)   lower case letters a b c d e f g h i j k l m n o p q r s t u v w x y z

11   (f)   other special characters ! $ % ? @ [\ ] ^ ` { } ~

12   Allowable replacements for the special characters vertical bar (|), sharp (#),
    and quotation (") are defined in section 2.10.

In DEC Ada on OpenVMS and Digital UNIX systems, each graphic character
corresponds to a unique code of the *ISO* eight-bit coded character set (*ISO*
standard 8859/1, commonly known as Latin-1).[2]

The characters included in each of the categories of basic graphic characters
are the *ISO* seven-bit characters plus the following additional characters. The
effect is to allow the use of the additional upper and lower case characters in
forming identifiers, and to allow the use of all the additional graphic characters
in forming character literals and string literals.

   (a)   upper case letters À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø
         Ù Ú Û Ü Ý Þ

   (b)   lower case letters ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü
         ý þ ÿ

   (c)   the nonbreaking space character and the soft hyphen

   (d)   other special characters ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼
         ½ ¾ ¿ × ÷

Notes:

13   The *ISO* character that corresponds to the sharp graphical symbol in the *ASCII*
    representation appears as a pound sterling symbol in the French, German, and
    United Kingdom standard national representations. In any case, the font
    design of graphical symbols (for example, whether they are in italic or bold
    typeface) is not part of the *ISO* standard.

---

[2]   See also Appendix G, AI-00866.

14 The meanings of the acronyms used in this section are as follows: *ANSI* stands for American National Standards Institute, *ASCII* stands for American Standard Code for Information Interchange, and *ISO* stands for International Organization for Standardization.

15 The following names are used when referring to special characters and other special characters:

16

symbol	name	symbol	name
"	quotation	>	greater than
#	sharp	_	underline
&	ampersand	|	vertical bar
'	apostrophe	!	exclamation mark
(	left parenthesis	$	dollar
)	right parenthesis	%	percent
*	star, multiply	?	question mark
+	plus	@	commercial at
,	comma	[	left square bracket
-	hyphen, minus	\	back-slash
.	dot, point, period	]	right square bracket
/	slash, divide	^	circumflex
:	colon	`	grave accent
;	semicolon	{	left brace
<	less than	}	right brace
=	equal	~	tilde

The following names are used when referring to the Latin-1 special characters:

symbol	name	symbol	name
¡	inverted exclamation mark	²	superscript two
¢	cent	³	superscript three
£	pound	´	acute accent
¤	currency	µ	micro
¥	yen	¶	pilcrow

symbol	name	symbol	name
¦	broken bar	·	middle dot
§	paragraph	¸	cedilla
¨	diaresis	¹	superscript one
©	copyright	º	masculine ordinal
ª	feminine ordinal	»	left angle quotation
«	right angle quotation	¼	one fourth
¬	not	½	one half
®	registered trademark	¾	three quarters
¯	macron	¿	inverted question mark
°	degree	×	times
±	plus or minus	÷	divide

2.2 Lexical Elements, Separators, and Delimiters

1   The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate lexical elements. Each lexical element is either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character literal, a string literal, or a comment. The effect of a program depends only on the particular sequences of lexical elements that form its compilations, excluding the comments, if any.

2   In some cases an explicit *separator* is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A separator is any of a space character, a format effector, or the end of a line. A space character is a separator except within a comment, a string literal, or a space character literal. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.

3   The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of a line is signified by one or more characters, then these characters must be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation must cause at least one end of line.

4   One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier or a numeric literal and an adjacent identifier or numeric literal.

5   A *delimiter* is either one of the following special characters (in the basic character set)

& ′ ( ) * + , − . / : ; < => |

6   or one of the following *compound delimiters* each composed of two adjacent special characters

=>   ..   **   :=   /=   >=   <=   <<   >>   <>

7   Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string literal, character literal, or numeric literal.

8   The remaining forms of lexical element are described in other sections of this chapter.

   Notes:

9   Each lexical element must fit on one line, since the end of a line is a separator. The quotation, sharp, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

10 The following names are used when referring to compound delimiters:

delimiter	name
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: "becomes")
/=	inequality (pronounced: "not equal")
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

11 **References:** character literal 2.5, comment 2.7, compilation 10.1, format effector 2.1, identifier 2.3, numeric literal 2.4, reserved word 2.9, space character 2.1, special character 2.1, string literal 2.6

2.3 Identifiers

1 Identifiers are used as names and also as reserved words.

```
2      identifier ::=
           letter {[underline] letter_or_digit}

       letter_or_digit ::= letter | digit

       letter ::= upper_case_letter | lower_case_letter
```

3 All characters of an identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same.

4 **Examples:**

```
COUNT     X     get_symbol    Ethelyn    Marion

SNOBOL_4  X1    PageCount      STORE_NEXT_ITEM
```

Note:

5 No space is allowed within an identifier since a space is a separator.

6 **References:** digit 2.1, lower case letter 2.1, name 4.1, reserved word 2.9, separator 2.2, space character 2.1, upper case letter 2.1

2.4 Numeric Literals

1 There are two classes of numeric literals: real literals and integer literals. A real literal is a numeric literal that includes a point; an integer literal is a numeric literal without a point. Real literals are the literals of the type *universal_real.* Integer literals are the literals of the type *universal_integer.*

2   numeric_literal ::= decimal_literal | based_literal

3 **References:** literal 4.2, universal_integer type 3.5.4, universal_real type 3.5.6

2.4.1 Decimal Literals

1   A decimal literal is a numeric literal expressed in the conventional decimal notation (that is, the base is implicitly ten).

2
```
decimal_literal ::= integer [.integer] [exponent]

integer  ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer
```

3   An underline character inserted between adjacent digits of a decimal literal does not affect the value of this numeric literal. The letter E of the exponent, if any, can be written either in lower case or in upper case, with the same meaning.

4   An exponent indicates the power of ten by which the value of the decimal literal without the exponent is to be multiplied to obtain the value of the decimal literal with the exponent. An exponent for an integer literal must not have a minus sign.

5   **Examples:**
```
12       0    1E6   123_456     --  integer literals

12.0     0.0  0.456 3.14159_26  --  real literals

1.34E-12 1.0E+6  --  real literals with exponent
```

Notes:

6   Leading zeros are allowed. No space is allowed in a numeric literal, not even between constituents of the exponent, since a space is a separator. A zero exponent is allowed for an integer literal.

7   **References:** digit 2.1, lower case letter 2.1, numeric literal 2.4, separator 2.2, space character 2.1, upper case letter 2.1

2.4.2 Based Literals

1   A based literal is a numeric literal expressed in a form that specifies the base explicitly. The base must be at least two and at most sixteen.

2
```
based_literal ::=
    base # based_integer [.based_integer] # [exponent]

base ::= integer

based_integer ::=
    extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter
```

3 An underline character inserted between adjacent digits of a based literal does not affect the value of this numeric literal. The base and the exponent, if any, are in decimal notation. The only letters allowed as extended digits are the letters A through F for the digits ten through fifteen. A letter in a based literal (either an extended digit or the letter E of an exponent) can be written either in lower case or in upper case, with the same meaning.

4 The conventional meaning of based notation is assumed; in particular the value of each extended digit of a based literal must be less than the base. An exponent indicates the power of the base by which the value of the based literal without the exponent is to be multiplied to obtain the value of the based literal with the exponent.[3]

5 **Examples:**
```
2#1111_1111#      16#FF#         016#0FF#      -- integer literals
                                               -- of value 255

16#E#E1           2#1110_0000#                 -- integer literals
                                               -- of value 224

16#F.FF#E+2       2#1.1111_1111_111#E11        -- real literals
                                               -- of value 4095.0
```

6 **References:** digit 2.1, exponent 2.4.1, letter 2.3, lower case letter 2.1, numeric literal 2.4, upper case letter 2.1

2.5 Character Literals

1 A character literal is formed by enclosing one of the 95 graphic characters (including the space) between two apostrophe characters. A character literal has a value that belongs to a character type.[4]

2 ```
character_literal ::= 'graphic_character'
```

By implementing characters as the *ISO* Latin-1 character set, DEC Ada also allows the additional 96 characters afforded by the eight-bit representation (see 2.1).[5]

3 **Examples:**
```
'A'   '*'   '''   ' '
```

4 **References:** character type 3.5.2, graphic character 2.1, literal 4.2, space character 2.1

---

[3] See also Appendix G, AI-00008.
[4] See also Appendix G, AI-00866.
[5] See also Appendix G, AI-00866.

## 2.6  String Literals

1   A string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation characters used as *string brackets.*

2       string_literal ::= "{graphic_character}"

3   A string literal has a value that is a sequence of character values corresponding to the graphic characters of the string literal apart from the quotation character itself.  If a quotation character value is to be represented in the sequence of character values, then a pair of adjacent quotation characters must be written at the corresponding place within the string literal.  (This means that a string literal that includes two adjacent quotation characters is never interpreted as two adjacent string literals.)

4   The *length* of a string literal is the number of character values in the sequence represented.  (Each doubled quotation character is counted as a single character.)

5   **Examples:**

```
"Message of the day:"

""                      --  an empty string literal
" "    "A"    """"      --  three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
```

   Note:

6   A string literal must fit on one line since it is a lexical element (see 2.2). Longer sequences of graphic character values can be obtained by catenation of string literals. Similarly catenation of constants declared in the package ASCII can be used to obtain sequences of character values that include nongraphic character values (the so-called control characters). Examples of such uses of catenation are given below:

```
"FIRST PART OF A SEQUENCE OF CHARACTERS " &
"THAT CONTINUES ON THE NEXT LINE"

"sequence that includes the" & ASCII.ACK & "control character"
```

7   **References:** ascii predefined package C, catenation operation 4.5.3, character value 3.5.2, constant 3.2.1, declaration 3.1, end of a line 2.2, graphic character 2.1, lexical element 2.2

2.7 Comments

1 A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a program. The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program; their sole purpose is the enlightenment of the human reader.[6]

2 **Examples:**

```
--  the last sentence above echoes the Algol 68 report

end; --  processing of LINE is complete

--  a long comment may be split onto
--  two or more consecutive lines

----------------  the first two hyphens start the comment
```

Note:

3 Horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces (see 2.2).

In DEC Ada, the set of characters allowed in a comment includes the graphic characters (see 2.1), the escape character, and the horizontal tabulation character.[7]

4 **References:** end of a line 2.2, illegal 1.6, legal 1.6, space character 2.1

2.8 Pragmas

1 A pragma is used to convey information to the compiler. A pragma starts with the reserved word **pragma** followed by an identifier that is the name of the pragma.

2
```
pragma ::=
    pragma identifier [(argument_association
                            {, argument_association})];

argument_association ::=
    [argument_identifier =>] name
  | [argument_identifier =>] expression
```

---

6 See also Appendix G, AI-00339 and AI-00866.
7 See also Appendix G, AI-00339 and AI-00866.

3　Pragmas are only allowed at the following places in a program:

4　• After a semicolon delimiter, but not within a formal part or discriminant part.[8]

5　• At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative", or one of the syntactic categories variant and exception handler; but not in place of such a construct. Also at any place where a compilation unit would be allowed.

6　Additional restrictions exist for the placement of specific pragmas.

7　Some pragmas have arguments. Argument associations can be either positional or named as for parameter associations of subprogram calls (see 6.4). Named associations are, however, only possible if the argument identifiers are defined. A name given in an argument must be either a name visible at the place of the pragma or an identifier specific to the pragma.

8　The pragmas defined by the language are described in Annex B: they must be supported by every implementation. In addition, an implementation may provide implementation-defined pragmas, which must then be described in Appendix F. An implementation is not allowed to define pragmas whose presence or absence influences the legality of the text outside such pragmas. Consequently, the legality of a program does not depend on the presence or absence of implementation-defined pragmas.[9]

9　A pragma that is not language-defined has no effect if its identifier is not recognized by the (current) implementation. Furthermore, a pragma (whether language-defined or implementation-defined) has no effect if its placement or its arguments do not correspond to what is allowed for the pragma. The region of text over which a pragma has an effect depends on the pragma.[10]

10　**Examples:**

```
pragma LIST(OFF);
pragma OPTIMIZE(TIME);
pragma INLINE(SETMASK);
pragma SUPPRESS(RANGE_CHECK, ON => INDEX);
```

---

[8]　See also Appendix G, AI-00388 and AI-00511.
[9]　 See also Appendix G, AI-00425 and AI-00511.
[10]　See also Appendix G, AI-00186, AI-00242, AI-00306, AI-00322, and AI-00371.

Note:

11   It is recommended (but not required) that implementations issue warnings for pragmas that are not recognized and therefore ignored.

12   **References:** compilation unit 10.1, delimiter 2.2, discriminant part 3.7.1, exception handler 11.2, expression 4.4, formal part 6.1, identifier 2.3, implementation-defined pragma F, language-defined pragma B, legal 1.6, name 4.1, reserved word 2.9, statement 5, static expression 4.9, variant 3.7.3, visibility 8.3

13   **Categories ending with "declaration" comprise:** basic declaration 3.1, component declaration 3.7, entry declaration 9.5, generic parameter declaration 12.1

14   **Categories ending with "clause" comprise:** alignment clause 13.4, component clause 13.4, context clause 10.1.1, representation clause 13.1, use clause 8.4, with clause 10.1.1

15   **Categories ending with "alternative" comprise:** accept alternative 9.7.1, case statement alternative 5.4, delay alternative 9.7.1, select alternative 9.7.1, selective wait alternative 9.7.1, terminate alternative 9.7.1

2.9 Reserved Words

1   The identifiers listed below are called *reserved words* and are reserved for special significance in the language. For readability of this manual, the reserved words appear in lower case boldface.

2

abort	**declare**	**generic**	**of**	**select**
abs	**delay**	**goto**	**or**	**separate**
accept	**delta**		**others**	**subtype**
access	**digits**	**if**	**out**	
all	**do**	**in**		**task**
and		**is**	**package**	**terminate**
array			**pragma**	**then**
at	**else**		**private**	**type**
	elsif	**limited**	**procedure**	
	end	**loop**		
begin	**entry**		**raise**	**use**
body	**exception**		**range**	
	exit	**mod**	**record**	**when**
			rem	**while**
		new	**renames**	**with**
case	**for**	**not**	**return**	
constant	**function**	**null**	**reverse**	**xor**

3   A reserved word must not be used as a declared identifier.

Notes:

4   Reserved words differing only in the use of corresponding upper and lower case letters are considered as the same (see 2.3). In some attributes the identifier that appears after the apostrophe is identical to some reserved word.

5   **References:** attribute 4.1.4, declaration 3.1, identifier 2.3, lower case letter 2.1, upper case letter 2.1

2.10 Allowable Replacements of Characters

1   The following replacements are allowed for the vertical bar, sharp, and quotation basic characters:

2   • A vertical bar character (|) can be replaced by an exclamation mark (!) where used as a delimiter.

3   • The sharp characters (#) of a based literal can be replaced by colons (:) provided that the replacement is done for both occurrences.

4   • The quotation characters (") used as string brackets at both ends of a string literal can be replaced by percent characters (%) provided that the enclosed sequence of characters contains no quotation character, and provided that both string brackets are replaced. Any percent character within the sequence of characters must then be doubled and each such doubled percent character is interpreted as a single percent character value.

In DEC Ada, the broken bar is not allowed as a replacement for the vertical bar.

5   These replacements do not change the meaning of the program.[11]

Notes:

6   It is recommended that use of the replacements for the vertical bar, sharp, and quotation characters be restricted to cases where the corresponding graphical symbols are not available. Note that the vertical bar appears as a broken bar on some equipment; replacement is not recommended in this case.

7   The rules given for identifiers and numeric literals are such that lower case and upper case letters can be used indifferently; these lexical elements can thus be written using only characters of the basic character set. If a string literal of the predefined type STRING contains characters that are not in the basic character set, the same sequence of character values can be obtained by catenating string literals that contain only characters of the basic character set with suitable character constants declared in the predefined package ASCII. Thus the string literal "AB$CD" could be replaced by "AB" & ASCII.DOLLAR & "CD". Similarly, the string literal "ABcd" with lower case letters could be replaced by "AB" & ASCII.LC_C & ASCII.LC_D.

8   **References:** ascii predefined package C, based literal 2.4.2, basic character 2.1, catenation operation 4.5.3, character value 3.5.2, delimiter 2.2, graphic character 2.1, graphical symbol 2.1, identifier 2.3, lexical element 2.2, lower case letter 2.1, numeric literal 2.4, string bracket 2.6, string literal 2.6, upper case letter 2.1

---

[11] See also Appendix G, AI-00350.

3

Declarations and Types

1 This chapter describes the types in the language and the rules for declaring constants, variables, and named numbers.

3.1 Declarations

1 The language defines several kinds of entities that are declared, either explicitly or implicitly, by declarations. Such an entity can be a numeric literal, an object, a discriminant, a record component, a loop parameter, an exception, a type, a subtype, a subprogram, a package, a task unit, a generic unit, a single entry, an entry family, a formal parameter (of a subprogram, entry, or generic subprogram), a generic formal parameter, a named block or loop, a labeled statement, or an operation (in particular, an attribute or an enumeration literal; see 3.3.3).

2 There are several forms of declaration. A basic declaration is a form of declaration defined as follows.

3
```
basic_declaration ::=
      object_declaration      | number_declaration
    | type_declaration        | subtype_declaration
    | subprogram_declaration  | package_declaration
    | task_declaration        | generic_declaration
    | exception_declaration   | generic_instantiation
    | renaming_declaration    | deferred_constant_declaration
```

4 Certain forms of declaration always occur (explicitly) as part of a basic declaration; these forms are discriminant specifications, component declarations, entry declarations, parameter specifications, generic parameter declarations, and enumeration literal specifications. A loop parameter specification is a form of declaration that occurs only in certain forms of loop statement.

5 The remaining forms of declaration are implicit: the name of a block, the name of a loop, and a statement label are implicitly declared. Certain operations are implicitly declared (see 3.3.3).

6   For each form of declaration the language rules define a certain region of text called the *scope* of the declaration (see 8.2). Several forms of declaration associate an identifier with a declared entity. Within its scope, and only there, there are places where it is possible to use the identifier to refer to the associated declared entity; these places are defined by the visibility rules (see 8.3). At such places the identifier is said to be a *name* of the entity (its simple name); the name is said to *denote* the associated entity.

7   Certain forms of enumeration literal specification associate a character literal with the corresponding declared entity. Certain forms of declaration associate an operator symbol or some other notation with an explicitly or implicitly declared operation.

8   The process by which a declaration achieves its effect is called the *elaboration* of the declaration; this process happens during program execution.

9   After its elaboration, a declaration is said to be *elaborated.* Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated. The elaboration of any declaration has always at least the effect of achieving this change of state (from not yet elaborated to elaborated). The phrase "*the elaboration has no other effect*" is used in this manual whenever this change of state is the only effect of elaboration for some form of declaration. An elaboration process is also defined for declarative parts, declarative items, and compilation units (see 3.9 and 10.5).

10   Object, number, type, and subtype declarations are described here. The remaining basic declarations are described in later chapters.

Note:

11   The syntax rules use the term *identifier* for the first occurrence of an identifier in some form of declaration; the term *simple name* is used for any occurrence of an identifier that already denotes some declared entity.

12   **References:** attribute 4.1.4, block name 5.6, block statement 5.6, character literal 2.5, component declaration 3.7, declarative item 3.9, declarative part 3.9, deferred constant declaration 7.4, discriminant specification 3.7.1, elaboration 3.9, entry declaration 9.5, enumeration literal specification 3.5.1, exception declaration 11.1, generic declaration 12.1, generic instantiation 12.3, generic parameter declaration 12.1, identifier 2.3, label 5.1, loop name 5.5, loop parameter specification 5.5, loop statement 5.5, name 4.1, number declaration 3.2.2, numeric literal 2.4, object declaration 3.2.1, operation 3.3, operator symbol 6.1, package declaration 7.1, parameter specification 6.1, record component 3.7, renaming declaration 8.5, representation clause 13.1, scope 8.2, simple name 4.1, subprogram body 6.3, subprogram declaration 6.1, subtype declaration 3.3.2, task declaration 9.1, type declaration 3.3.1, visibility 8.3

3.2 Objects and Named Numbers

1   An *object* is an entity that contains (has) a value of a given type. An object is one of the following:

2   • an object declared by an object declaration or by a single task declaration,

3   • a formal parameter of a subprogram, entry, or generic subprogram,

4   • a generic formal object,

5   • a loop parameter,

6   • an object designated by a value of an access type,

7   • a component or a slice of another object.

8   A number declaration is a special form of object declaration that associates an identifier with a value of type *universal_integer* or *universal_real.* [1]

9
```
object_declaration ::=
    identifier_list : [constant] subtype_indication
                          [:= expression];
  | identifier_list : [constant] constrained_array_definition
                          [:= expression];

number_declaration ::=
    identifier_list : constant := universal_static_expression;

identifier_list ::=  identifier {, identifier}
```

10 An object declaration is called a *single object declaration* if its identifier list has a single identifier; it is called a *multiple object declaration* if the identifier list has two or more identifiers. A multiple object declaration is equivalent to a sequence of the corresponding number of single object declarations. For each identifier of the list, the equivalent sequence has a single object declaration formed by this identifier, followed by a colon and by whatever appears at the right of the colon in the multiple object declaration; the equivalent sequence is in the same order as the identifier list.

11 A similar equivalence applies also for the identifier lists of number declarations, component declarations, discriminant specifications, parameter specifications, generic parameter declarations, exception declarations, and deferred constant declarations.

12 In the remainder of this reference manual, explanations are given for declarations with a single identifier; the corresponding explanations for declarations with several identifiers follow from the equivalence stated above.

---

[1] See also Appendix G, AI-00263.

13 **Example:**
```
--  the multiple object declaration

JOHN, PAUL : PERSON_NAME := new PERSON(SEX => M);  --  see 3.8.1

--  is equivalent to the two single object declarations
--  in the order given

JOHN : PERSON_NAME := new PERSON(SEX => M);
PAUL : PERSON_NAME := new PERSON(SEX => M);
```

14 **References:** access type 3.8, constrained array definition 3.6, component 3.3, declaration 3.1, deferred constant declaration 7.4, designate 3.8, discriminant specification 3.7.1, entry 9.5, exception declaration 11.1, expression 4.4, formal parameter 6.1, generic formal object 12.1.1, generic parameter declaration 12.1, generic unit 12, generic subprogram 12.1, identifier 2.3, loop parameter 5.5, numeric type 3.5, parameter specification 6.1, scope 8.2, simple name 4.1, single task declaration 9.1, slice 4.1.2, static expression 4.9, subprogram 6, subtype indication 3.3.2, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6

3.2.1 Object Declarations

1 An object declaration declares an object whose type is given either by a subtype indication or by a constrained array definition. If the object declaration includes the assignment compound delimiter followed by an expression, the expression specifies an initial value for the declared object; the type of the expression must be that of the object.

2 The declared object is a *constant* if the reserved word **constant** appears in the object declaration; the declaration must then include an explicit initialization. The value of a constant cannot be modified after initialization. Formal parameters of mode **in** of subprograms and entries, and generic formal parameters of mode **in**, are also constants; a loop parameter is a constant within the corresponding loop; a subcomponent or slice of a constant is a constant.

3 An object that is not a constant is called a *variable* (in particular, the object declared by an object declaration that does not include the reserved word **constant** is a variable). The only ways to change the value of a variable are either directly by an assignment, or indirectly when the variable is updated (see 6.2) by a procedure or entry call statement (this action can be performed either on the variable itself, on a subcomponent of the variable, or on another variable that has the given variable as subcomponent).

4 The elaboration of an object declaration proceeds as follows:

5   (a) The subtype indication or the constrained array definition is first elaborated. This establishes the subtype of the object.

6   (b)   If the object declaration includes an explicit initialization, the initial value is obtained by evaluating the corresponding expression. Otherwise any implicit initial values for the object or for its subcomponents are evaluated.

7   (c)   The object is created.

8   (d)   Any initial value (whether explicit or implicit) is assigned to the object or to the corresponding subcomponent.

9   Implicit initial values are defined for objects declared by object declarations, and for components of such objects, in the following cases:

10   •   If the type of an object is an access type, the implicit initial value is the null value of the access type.

11   •   If the type of an object is a task type, the implicit initial (and only) value designates a corresponding task.

12   •   If the type of an object is a type with discriminants and the subtype of the object is constrained, the implicit initial (and only) value of each discriminant is defined by the subtype of the object.

13   •   If the type of an object is a composite type, the implicit initial value of each component that has a default expression is obtained by evaluation of this expression, unless the component is a discriminant of a constrained object (the previous case).

14   In the case of a component that is itself a composite object and whose value is defined neither by an explicit initialization nor by a default expression, any implicit initial values for components of the composite object are defined by the same rules as for a declared object.

15   The steps (a) to (d) are performed in the order indicated. For step (b), if the default expression for a discriminant is evaluated, then this evaluation is performed before that of default expressions for subcomponents that depend on discriminants, and also before that of default expressions that include the name of the discriminant. Apart from the previous rule, the evaluation of default expressions is performed in some order that is not defined by the language.

16   The initialization of an object (the declared object or one of its subcomponents) checks that the initial value belongs to the subtype of the object; for an array object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement, unless the object is a constant whose

subtype is an unconstrained array type. The exception CONSTRAINT_ERROR is raised if this check fails. [2]

17   The value of a scalar variable is undefined after elaboration of the corresponding object declaration unless an initial value is assigned to the variable by an initialization (explicitly or implicitly).

18   If the operand of a type conversion or qualified expression is a variable that has scalar subcomponents with undefined values, then the values of the corresponding subcomponents of the result are undefined. The execution of a program is erroneous if it attempts to evaluate a scalar variable with an undefined value. Similarly, the execution of a program is erroneous if it attempts to apply a predefined operator to a variable that has a scalar subcomponent with an undefined value. [3]

19   **Examples of variable declarations:**
```
COUNT, SUM  : INTEGER;
SIZE        : INTEGER range 0 .. 10_000 := 0;
SORTED      : BOOLEAN := FALSE;
COLOR_TABLE : array(1 .. N) of COLOR;
OPTION      : BIT_VECTOR(1 .. 10) := (others => TRUE);
```

20   **Examples of constant declarations:**
```
LIMIT     : constant INTEGER := 10_000;
LOW_LIMIT : constant INTEGER := LIMIT/10;
TOLERANCE : constant REAL := DISPERSION(1.15);
```

Note:

21   The expression initializing a constant object need not be a static expression (see 4.9). In the above examples, LIMIT and LOW_LIMIT are initialized with static expressions, but TOLERANCE is not if DISPERSION is a user-defined function.

22   **References:** access type 3.8, assignment 5.2, assignment compound delimiter 5.2, component 3.3, composite type 3.3, constrained array definition 3.6, constrained subtype 3.3, constraint_error exception 11.1, conversion 4.6, declaration 3.1, default expression for a discriminant 3.7, default initial value for an access type 3.8, depend on a discriminant 3.7.1, designate 3.8, discriminant 3.3, elaboration 3.9, entry 9.5, evaluation 4.5, expression 4.4, formal parameter 6.1, generic formal parameter 12.1 12.3, generic unit 12, in some order 1.6, limited type 7.4.4, mode in 6.1, package 7, predefined operator 4.5, primary 4.4, private type 7.4, qualified expression 4.7, reserved word 2.9, scalar type 3.5, slice 4.1.2, subcomponent 3.3, subprogram 6, subtype 3.3, subtype indication 3.3.2, task 9, task type 9.2, type 3.3, visible part 7.2

---

[2]   See also Appendix G, AI-00308.
[3]   See also Appendix G,AI-00155, AI-00356, AI-00374, and AI-00426.

3.2.2 Number Declarations

1   A number declaration is a special form of constant declaration. The type of the static expression given for the initialization of a number declaration must be either the type *universal_integer* or the type *universal_real.* The constant declared by a number declaration is called a *named number* and has the type of the static expression.

Note:

2   The rules concerning expressions of a universal type are explained in section 4.10. It is a consequence of these rules that if every primary contained in the expression is of the type *universal_integer*, then the named number is also of this type. Similarly, if every primary is of the type *universal_real*, then the named number is also of this type.

3   **Examples of number declarations:**

```
PI             : constant := 3.14159_26536; -- a real number
TWO_PI         : constant := 2.0*PI;        -- a real number
MAX            : constant := 500;           -- an integer number
POWER_16       : constant := 2**16;         -- the integer 65_536
ONE, UN, EINS  : constant := 1;             -- three different
                                            -- names for 1
```

4   **References:** identifier 2.3, primary 4.4, static expression 4.9, type 3.3, universal_ integer type 3.5.4, universal_real type 3.5.6, universal type 4.10

3.3 Types and Subtypes

1   A type is characterized by a set of values and a set of operations.

2   There exist several *classes* of types. *Scalar* types are integer types, real types, and types defined by enumeration of their values; values of these types have no components. *Array* and *record* types are composite; a value of a composite type consists of *component* values. An *access* type is a type whose values provide access to objects. *Private* types are types for which the set of possible values is well defined, but not directly available to the users of such types. Finally, there are *task* types. (Private types are described in chapter 7, task types are described in chapter 9, the other classes of types are described in this chapter.)

3   Certain record and private types have special components called *discriminants* whose values distinguish alternative forms of values of one of these types. If a private type has discriminants, they are known to users of the type. Hence a private type is only known by its name, its discriminants if any, and by the corresponding set of operations.

4   The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case where the constraint imposes no restriction is also included); a value is said to *satisfy* a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint; a value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained*: it corresponds to a condition that imposes no restriction. The base type of a type is the type itself.

5   The set of operations defined for a subtype of a given type includes the operations that are defined for the type; however the assignment operation to a variable having a given subtype only assigns values that belong to the subtype. Additional operations, such as qualification (in a qualified expression), are implicitly defined by a subtype declaration.

6   Certain types have *default initial values* defined for objects of the type; certain other types have *default expressions* defined for some or all of their components. Certain operations of types and subtypes are called *attributes*; these operations are denoted by the form of name described in section 4.1.4.

7   The term *subcomponent* is used in this manual in place of the term component to indicate either a component, or a component of another component or subcomponent. Where other subcomponents are excluded, the term component is used instead.

8   A given type must not have a subcomponent whose type is the given type itself.

9   The name of a class of types is used in this manual as a qualifier for objects and values that have a type of the class considered. For example, the term "array object" is used for an object whose type is an array type; similarly, the term "access value" is used for a value of an access type.

Note:

10 The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset; it can be an empty subset.

11 **References:** access type 3.8, array type 3.6, assignment 5.2, attribute 4.1.4, component of an array 3.6, component of a record 3.7, discriminant constraint 3.7.2, enumeration type 3.5.1, integer type 3.5.4, object 3.2.1, private type 7.4, qualified expression 4.7, real type 3.5.6, record type 3.7, subtype declaration 3.3.2, task type 9.1, type declaration 3.3.1

3.3.1 Type Declarations

1   A type declaration declares a type.

2
```
type_declaration ::=  full_type_declaration
    | incomplete_type_declaration | private_type_declaration

full_type_declaration ::=
    type identifier [discriminant_part] is type_definition;

type_definition ::=
    enumeration_type_definition | integer_type_definition
    | real_type_definition         | array_type_definition
    | record_type_definition       | access_type_definition
    | derived_type_definition
```

3   The elaboration of a full type declaration consists of the elaboration of the discriminant part, if any (except in the case of the full type declaration for an incomplete or private type declaration), and of the elaboration of the type definition.

4   The types created by the elaboration of distinct type definitions are distinct types. Moreover, the elaboration of the type definition for a numeric or derived type creates both a base type and a subtype of the base type; the same holds for a constrained array definition (one of the two forms of array type definition).

5   The simple name declared by a full type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype, and the base type is anonymous. A type is said to be *anonymous* if it has no simple name. For explanatory purposes, this reference manual sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier.

6   **Examples of type definitions:**
```
(WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK)
range 1 .. 72
array(1 .. 10) of INTEGER
```

7   **Examples of type declarations:**
```
type COLOR  is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type COLUMN is range 1 .. 72;
type TABLE  is array(1 .. 10) of INTEGER;
```

Notes:

⁸ Two type definitions always define two distinct types, even if they are textually identical. Thus, the array type definitions given in the declarations of A and B below define distinct types.

```
A : array(1 .. 10) of BOOLEAN;
B : array(1 .. 10) of BOOLEAN;
```

⁹ If A and B are declared by a multiple object declaration as below, their types are nevertheless different, since the multiple object declaration is equivalent to the above two single object declarations.

```
A, B : array(1 .. 10) of BOOLEAN;
```

¹⁰ Incomplete type declarations are used for the definition of recursive and mutually dependent types (see 3.8.1). Private type declarations are used in package specifications and in generic parameter declarations (see 7.4 and 12.1).

¹¹ **References:** access type definition 3.8, array type definition 3.6, base type 3.3, constrained array definition 3.6, constrained subtype 3.3, declaration 3.1, derived type 3.4, derived type definition 3.4, discriminant part 3.7.1, elaboration 3.9, enumeration type definition 3.5.1, identifier 2.3, incomplete type declaration 3.8.1, integer type definition 3.5.4, multiple object declaration 3.2, numeric type 3.5, private type declaration 7.4, real type definition 3.5.6, reserved word 2.9, type 3.3

3.3.2 Subtype Declarations

¹ A subtype declaration declares a subtype.

```
2    subtype_declaration ::=
         subtype identifier is subtype_indication;

     subtype_indication ::=  type_mark [constraint]

     type_mark ::= type_name | subtype_name

     constraint ::=
         range_constraint         | floating_point_constraint
       | fixed_point_constraint | index_constraint
       | discriminant_constraint
```

³ A type mark denotes a type or a subtype. If a type mark is the name of a type, the type mark denotes this type and also the corresponding unconstrained subtype. The *base type of a type mark* is, by definition, the base type of the type or subtype denoted by the type mark.

⁴ A subtype indication defines a subtype of the base type of the type mark.

5 If an index constraint appears after a type mark in a subtype indication, the type mark must not already impose an index constraint. Likewise for a discriminant constraint, the type mark must not already impose a discriminant constraint.

6 The elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows: [4]

7   (a)   The constraint is first elaborated.

8   (b)   A check is then made that the constraint is *compatible* with the type or subtype denoted by the type mark.

9 The condition imposed by a constraint is the condition obtained after elaboration of the constraint. (The rules of constraint elaboration are such that the expressions and ranges of constraints are evaluated by the elaboration of these constraints.) The rules defining compatibility are given for each form of constraint in the appropriate section. These rules are such that if a constraint is compatible with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. The exception CONSTRAINT_ERROR is raised if any check of compatibility fails.

10 **Examples of subtype declarations:**
```
subtype RAINBOW    is COLOR range RED .. BLUE;  -- see 3.3.1
subtype RED_BLUE  is RAINBOW;
subtype INT        is INTEGER;
subtype SMALL_INT is INTEGER range -10 .. 10;
subtype UP_TO_K    is COLUMN range 1 .. K;       -- see 3.3.1
subtype SQUARE     is MATRIX(1 .. 10, 1 .. 10); -- see 3.6
subtype MALE       is PERSON(SEX => M);          -- see 3.8
```

Note:

11 A subtype declaration does not define a new type.

12 **References:** base type 3.3, compatibility of discriminant constraints 3.7.2, compatibility of fixed point constraints 3.5.9, compatibility of floating point constraints 3.5.7, compatibility of index constraints 3.6.1, compatibility of range constraints 3.5, constraint_error exception 11.1, declaration 3.1, discriminant 3.3, discriminant constraint 3.7.2, elaboration 3.9, evaluation 4.5, expression 4.4, floating point constraint 3.5.7, fixed point constraint 3.5.9, index constraint 3.6.1, range constraint 3.5, reserved word 2.9, subtype 3.3, type 3.3, type name 3.3.1, unconstrained subtype 3.3

---

[4]   See also Appendix G, AI-00449.

3.3.3 Classification of Operations

1   The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type; such subprograms are necessarily declared after the type declaration. [5]

2   The remaining operations are each implicitly declared for a given type declaration, immediately after the type definition. These implicitly declared operations comprise the *basic* operations, the predefined operators (see 4.5), and enumeration literals. In the case of a derived type declaration, the implicitly declared operations include any derived subprograms. The operations implicitly declared for a given type declaration occur after the type declaration and before the next explicit declaration, if any. The implicit declarations of derived subprograms occur last.

3   A basic operation is an operation that is inherent in one of the following:

4   • An assignment (in assignment statements and initializations), an allocator, a membership test, or a short-circuit control form.

5   • A selected component, an indexed component, or a slice.

6   • A qualification (in qualified expressions), an explicit type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type.

7   • A numeric literal (for a universal type), the literal **null** (for an access type), a string literal, an aggregate, or an attribute.

8   For every type or subtype T, the following attribute is defined:

9   T′BASE   The base type of T. This attribute is allowed only as the prefix of the name of another attribute: for example, T′BASE′FIRST.

Note:

10   Each literal is an operation whose evaluation yields the corresponding value (see 4.2). Likewise, an aggregate is an operation whose evaluation yields a value of a composite type (see 4.3). Some operations of a type *operate on* values of the type, for example, predefined operators and certain subprograms and attributes. The evaluation of some operations of a type *returns* a value of the type, for example, literals and certain functions, attributes, and predefined operators. Assignment is an operation that operates on an object and a value. The evaluation of the operation corresponding to a selected component, an indexed component, or a slice, yields the object or value denoted by this form of name.

---

[5]   See also Appendix G, AI-00330.

11 **References:** aggregate 4.3, allocator 4.8, assignment 5.2, attribute 4.1.4, character literal 2.5, composite type 3.3, conversion 4.6, derived subprogram 3.4, enumeration literal 3.5.1, formal parameter 6.1, function 6.5, indexed component 4.1.1, initial value 3.2.1, literal 4.2, membership test 4.5 4.5.2, null literal 3.8, numeric literal 2.4, numeric type 3.5, object 3.2.1, 6.1, predefined operator 4.5, qualified expression 4.7, selected component 4.1.3, short-circuit control form 4.5 4.5.1, slice 4.1.2, string literal 2.6, subprogram 6, subtype 3.3, type 3.3, type declaration 3.3.1, universal_integer type 3.5.4, universal_real type 3.5.6, universal type 4.10

3.4 Derived Types

1 A derived type definition defines a new (base) type whose characteristics are derived from those of a *parent type*; the new type is called a *derived type.* A derived type definition further defines a *derived subtype*, which is a subtype of the derived type.

2   derived_type_definition ::= **new** subtype_indication

3 The subtype indication that occurs after the reserved word **new** defines the *parent subtype.* The parent type is the base type of the parent subtype. If a constraint exists for the parent subtype, a similar constraint exists for the derived subtype; the only difference is that for a range constraint, and likewise for a floating or fixed point constraint that includes a range constraint, the value of each bound is replaced by the corresponding value of the derived type. The characteristics of the derived type are defined as follows:

4   • The derived type belongs to the same class of types as the parent type. The set of possible values for the derived type is a copy of the set of possible values for the parent type. If the parent type is composite, then the same components exist for the derived type, and the subtype of corresponding components is the same.

5   • For each basic operation of the parent type, there is a corresponding basic operation of the derived type. Explicit type conversion of a value of the parent type into the corresponding value of the derived type is allowed and vice versa as explained in section 4.6.

6   • For each enumeration literal or predefined operator of the parent type there is a corresponding operation for the derived type.

7   • If the parent type is a task type, then for each entry of the parent type there is a corresponding entry for the derived type.

8   • If a default expression exists for a component of an object having the parent type, then the same default expression is used for the corresponding component of an object having the derived type.

9 • If the parent type is an access type, then the parent and the derived type share the same collection; there is a null access value for the derived type and it is the default initial value of that type.

10 • If an explicit representation clause exists for the parent type and if this clause appears before the derived type definition, then there is a corresponding representation clause (an implicit one) for the derived type. [6]

11 • Certain subprograms that are operations of the parent type are said to be *derivable.* For each derivable subprogram of the parent type, there is a corresponding derived subprogram for the derived type. Two kinds of derivable subprograms exist. First, if the parent type is declared immediately within the visible part of a package, then a subprogram that is itself explicitly declared immediately within the visible part becomes derivable after the end of the visible part, if it is an operation of the parent type. (The explicit declaration is by a subprogram declaration, a renaming declaration, or a generic instantiation.) Second, if the parent type is itself a derived type, then any subprogram that has been derived by this parent type is further derivable, unless the parent type is declared in the visible part of a package and the derived subprogram is hidden by a derivable subprogram of the first kind. [7]

12 Each operation of the derived type is implicitly declared at the place of the derived type declaration. The implicit declarations of any derived subprograms occur last.

13 The specification of a derived subprogram is obtained implicitly by systematic replacement of the parent type by the derived type in the specification of the derivable subprogram. Any subtype of the parent type is likewise replaced by a subtype of the derived type with a similar constraint (as for the transformation of a constraint of the parent subtype into the corresponding constraint of the derived subtype). Finally, any expression of the parent type is made to be the operand of a type conversion that yields a result of the derived type.

14 Calling a derived subprogram is equivalent to calling the corresponding subprogram of the parent type, in which each actual parameter that is of the derived type is replaced by a type conversion of this actual parameter to the parent type (this means that a conversion to the parent type happens before the call for the modes **in** and **in out**; a reverse conversion to the derived type happens after the call for the modes **in out** and **out**, see 6.4.1). In addition, if

---

[6] See also Appendix G, AI-00138 and AI-00292.
[7] See also Appendix G, AI-00367 and AI-00398.

the result of a called function is of the parent type, this result is converted to the derived type.

15   If a derived or private type is declared immediately within the visible part of a package, then, within this visible part, this type must not be used as the parent type of a derived type definition. (For private types, see also section 7.4.1.)

16   For the elaboration of a derived type definition, the subtype indication is first elaborated, the derived type is then created, and finally, the derived subtype is created.

17   **Examples:**

```
type LOCAL_COORDINATE is new COORDINATE;  --  two different types
type MIDWEEK is new DAY range TUE .. THU; --  see 3.5.1
type COUNTER is new POSITIVE;            --  same range as
                                         -- POSITIVE

type SPECIAL_KEY is new KEY_MANAGER.KEY;  -- see 7.4.2
-- the derived subprograms have the following specifications:

-- procedure GET_KEY(K : out SPECIAL_KEY);
-- function "<"(X,Y : SPECIAL_KEY) return BOOLEAN;
```

Notes:

18   The rules of derivation of basic operations and enumeration literals imply that the notation for any literal or aggregate of the derived type is the same as for the parent type; such literals and aggregates are said to be *overloaded.* Similarly, it follows that the notation for denoting a component, a discriminant, an entry, a slice, or an attribute is the same for the derived type as for the parent type.

19   Hiding of a derived subprogram is allowed even within the same declarative region (see 8.3). A derived subprogram hides a predefined operator that has the same parameter and result type profile (see 6.6).

20   A generic subprogram declaration is not derivable since it declares a generic unit rather than a subprogram. On the other hand, an instantiation of a generic subprogram is a (nongeneric) subprogram, which is derivable if it satisfies the requirements for derivability of subprograms.

21   If the parent type is a boolean type, the predefined relational operators of the derived type deliver a result of the predefined type BOOLEAN (see 4.5.2).

22   If a representation clause is given for the parent type but appears after the derived type declaration, then no corresponding representation clause applies

to the derived type; hence an explicit representation clause for such a derived type is allowed. [8]

23 For a derived subprogram, if a parameter belongs to the derived type, the subtype of this parameter need not have any value in common with the derived subtype.

24 **References:** access value 3.8, actual parameter 6.4.1, aggregate 4.3, attribute 4.1.4, base type 3.3, basic operation 3.3.3, boolean type 3.5.3, bound of a range 3.5, class of type 3.3, collection 3.8, component 3.3, composite type 3.3, constraint 3.3, conversion 4.6, declaration 3.1, declarative region 8.1, default expression 3.2.1, default initial value for an access type 3.8, discriminant 3.3, elaboration 3.9, entry 9.5, enumeration literal 3.5.1, floating point constraint 3.5.7, fixed point constraint 3.5.9, formal parameter 6.1, function call 6.4, generic declaration 12.1, immediately within 8.1, implicit declaration 3.1, literal 4.2, mode 6.1, overloading 6.6 8.7, package 7, package specification 7.1, parameter association 6.4, predefined operator 4.5, private type 7.4, procedure 6, procedure call statement 6.4, range constraint 3.5, representation clause 13.1, reserved word 2.9, slice 4.1.2, subprogram 6, subprogram specification 6.1, subtype indication 3.3.2, subtype 3.3, type 3.3, type definition 3.3.1, visible part 7.2

3.5 Scalar Types

1 Scalar types comprise enumeration types, integer types, and real types. Enumeration types and integer types are called *discrete* types; each value of a discrete type has a position number which is an integer value. Integer types and real types are called *numeric* types. All scalar types are ordered, that is, all relational operators are predefined for their values.

2
```
range_constraint ::=  range range

range ::=  range_attribute
    | simple_expression .. simple_expression
```

3 A range specifies a subset of values of a scalar type. The range L .. R specifies the values from L to R inclusive if the relation L <= R is true. The values L and R are called the *lower bound* and *upper bound* of the range, respectively. A value V is said to *satisfy* a range constraint if it belongs to the range; the value V is said to *belong* to the range if the relations L <= V and V <= R are both TRUE. A *null* range is a range for which the relation R < L is TRUE; no value belongs to a null range. The operators <= and < in the above definitions are the predefined operators of the scalar type.

---

[8] See also Appendix G, AI-00138.

4 If a range constraint is used in a subtype indication, either directly or as part of a floating or fixed point constraint, the type of the simple expressions (likewise, of the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype, or if the range constraint defines a null range; otherwise the range constraint is not compatible with the subtype.

5 The elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines its lower bound and its upper bound. If simple expressions are given to specify the bounds, the evaluation of the range evaluates these simple expressions in some order that is not defined by the language.

6 **Attributes:**

7 For any scalar type T or for any subtype T of a scalar type, the following attributes are defined:

8 T′FIRST   Yields the lower bound of T. The value of this attribute has the same type as T.

9 T′LAST   Yields the upper bound of T. The value of this attribute has the same type as T.

Note:

10 Indexing and iteration rules use values of discrete types.

11 **References:** attribute 4.1.4, constraint 3.3, enumeration type 3.5.1, erroneous 1.6, evaluation 4.5, fixed point constraint 3.5.9, floating point constraint 3.5.7, index 3.6, integer type 3.5.4, loop statement 5.5, range attribute 3.6.2, real type 3.5.6, relational operator 4.5 4.5.2, satisfy a constraint 3.3, simple expression 4.4, subtype indication 3.3.2, type mark 3.3.2

3.5.1 Enumeration Types

1 An enumeration type definition defines an enumeration type.

2
```
enumeration_type_definition ::=
    (enumeration_literal_specification
        {, enumeration_literal_specification})

enumeration_literal_specification ::=  enumeration_literal

enumeration_literal ::=  identifier | character_literal
```

3   The identifiers and character literals listed by an enumeration type definition must be distinct. Each enumeration literal specification is the declaration of the corresponding enumeration literal: this declaration is equivalent to the declaration of a parameterless function, the designator being the enumeration literal, and the result type being the enumeration type. The elaboration of an enumeration type definition creates an enumeration type; this elaboration includes that of every enumeration literal specification. [9]

4   Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each other enumeration literal is one more than for its predecessor in the list.

5   If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding literals are said to be *overloaded.* At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal must be determinable from the context (see 8.7).

6   **Examples:**
```
type DAY    is (MON, TUE, WED, THU, FRI, SAT, SUN);
type SUIT   is (CLUBS, DIAMONDS, HEARTS, SPADES);
type GENDER is (M, F);
type LEVEL  is (LOW, MEDIUM, URGENT);
type COLOR  is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type LIGHT  is (RED, AMBER, GREEN); -- RED and GREEN are
                                    -- overloaded

type HEXA   is ('A', 'B', 'C', 'D', 'E', 'F');
type MIXED  is ('A', 'B', '*', B, NONE, '?', '%');

subtype WEEKDAY is DAY   range MON .. FRI;
subtype MAJOR   is SUIT  range HEARTS .. SPADES;
subtype RAINBOW is COLOR range RED .. BLUE;  --  the color RED,
                                             --  not the light
```

Note:

7   If an enumeration literal occurs in a context that does not otherwise suffice to determine the type of the literal, then qualification by the name of the enumeration type is one way to resolve the ambiguity (see 8.7).

8   **References:** character literal 2.5, declaration 3.1, designator 6.1, elaboration 3.9, 6.1, function 6.5, identifier 2.3, name 4.1, overloading 6.6 8.7, position number 3.5, qualified expression 4.7, relational operator 4.5 4.5.2, type 3.3, type definition 3.3.1

---

[9]   See also Appendix G, AI-00330 and AI-00430.

3.5.2 Character Types

1   An enumeration type is said to be a character type if at least one of its enumeration literals is a character literal. The predefined type CHARACTER is a character type whose values are the 128 characters of the *ASCII* character set. Each of the 95 graphic characters of this character set is denoted by the corresponding character literal.

2   **Example:**

```
type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

Notes:

3   The predefined package ASCII includes the declaration of constants denoting control characters and of constants denoting graphic characters that are not in the basic character set.

4   A conventional character set such as *EBCDIC* can be declared as a character type; the internal codes of the characters can be specified by an enumeration representation clause as explained in section 13.3.

5   **References:** ascii predefined package C, basic character 2.1, character literal 2.5, constant 3.2.1, declaration 3.1, enumeration type 3.5.1, graphic character 2.1, identifier 2.3, literal 4.2, predefined type C, type 3.3

3.5.3 Boolean Types

1   There is a predefined enumeration type named BOOLEAN. It contains the two literals FALSE and TRUE ordered with the relation FALSE < TRUE. A boolean type is either the type BOOLEAN or a type that is derived, directly or indirectly, from a boolean type.

2   **References:** derived type 3.4, enumeration literal 3.5.1, enumeration type 3.5.1, relational operator 4.5 4.5.2, type 3.3

3.5.4 Integer Types

1   An integer type definition defines an integer type whose set of values includes at least those of the specified range.

2   `integer_type_definition ::= range_constraint`

3   If a range constraint is used as an integer type definition, each bound of the range must be defined by a static expression of some integer type, but the two bounds need not have the same integer type.
(Negative bounds are allowed.) [10]

---

[10] See also Appendix G, AI-00240.

4   A type declaration of the form: [11]

```
type T is range L .. R;
```

5   is, by definition, equivalent to the following declarations:

```
type integer_type is new predefined_integer_type;
subtype T is integer_type
   range integer_type(L) .. integer_type(R);
```

6   where *integer_type* is an anonymous type, and where the predefined integer type is implicitly selected by the implementation, so as to contain the values L to R inclusive. The integer type declaration is illegal if none of the predefined integer types satisfies this requirement, excepting *universal_integer.* The elaboration of the declaration of an integer type consists of the elaboration of the equivalent type and subtype declarations.

7   The predefined integer types include the type INTEGER. An implementation may also have predefined types such as SHORT_INTEGER and LONG_INTEGER, which have (substantially) shorter and longer ranges, respectively, than INTEGER. The range of each of these types must be symmetric about zero, excepting an extra negative value which may exist in some implementations. The base type of each of these types is the type itself.

DEC Ada provides the following predefined integer types. Additional integer types are declared in the package SYSTEM. See section 13.7a.4 for more information.

Predefined type	Range of values	DEC Ada systems on which it applies
LONG_INTEGER	-2^{31} .. $2^{31}-1$ (or $-2,147,483,648$ .. $2,147,483,647$)	OpenVMS VAX
	-2^{63} .. $2^{63}-1$	Alpha
INTEGER	-2^{31} .. $2^{31}-1$ (or $-2,147,483,648$ .. $2,147,483,647$)	All
SHORT_INTEGER	-2^{15} .. $2^{15}-1$ (or $-32,768$ .. $32,767$)	All
SHORT_SHORT_INTEGER	-2^{7} .. $2^{7}-1$ (or -128 .. 127)	All

8   Integer literals are the literals of an anonymous predefined integer type that is called *universal_integer* in this reference manual. Other integer types have no literals. However, for each integer type there exists an implicit conversion that

---

[11] See also Appendix G, AI-00023.

converts a *universal_integer* value into the corresponding value (if any) of the integer type. The circumstances under which these implicit conversions are invoked are described in section 4.6.

9   The position number of an integer value is the corresponding value of the type *universal_integer.*

10   The same arithmetic operators are predefined for all integer types (see 4.5). The exception NUMERIC_ERROR is raised by the execution of an operation (in particular an implicit conversion) that cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type). However, an implementation is not required to raise the exception NUMERIC_ERROR if the operation is part of a larger expression whose result can be computed correctly, as described in section 11.6. [12]

11   **Examples:**

```
type PAGE_NUM  is range 1 .. 2_000;
type LINE_SIZE is range 1 .. MAX_LINE_SIZE;

subtype SMALL_INT   is INTEGER   range -10 .. 10;
subtype COLUMN_PTR  is LINE_SIZE range 1 .. 10;
subtype BUFFER_SIZE is INTEGER   range 0 .. MAX;
```

   Notes:

12   The name declared by an integer type declaration is a subtype name. On the other hand, the predefined operators of an integer type deliver results whose range is defined by the parent predefined type; such a result need not belong to the declared subtype, in which case an attempt to assign the result to a variable of the integer subtype raises the exception CONSTRAINT_ERROR.

13   The smallest (most negative) value supported by the predefined integer types of an implementation is the named number SYSTEM.MIN_INT and the largest (most positive) value is SYSTEM.MAX_INT (see 13.7).

14   **References:** anonymous type 3.3.1, belong to a subtype 3.3, bound of a range 3.5, constraint_error exception 11.1, conversion 4.6, identifier 2.3, integer literal 2.4, literal 4.2, numeric_error exception 11.1, parent type 3.4, predefined operator 4.5, range constraint 3.5, static expression 4.9, subtype declaration 3.3.2, system predefined package 13.7, type 3.3, type declaration 3.3.1, type definition 3.3.1, universal type 4.10

─────────────────────

[12] See also Appendix G, AI-00267 and AI-00387.

3.5.5 Operations of Discrete Types

1   The basic operations of a discrete type include the operations involved in assignment, the membership tests, and qualification; for a boolean type they include the short-circuit control forms; for an integer type they include the explicit conversion of values of other numeric types to the integer type, and the implicit conversion of values of the type *universal_integer* to the type.

2   Finally, for every discrete type or subtype T, the basic operations include the attributes listed below. In this presentation, T is referred to as being a subtype (the subtype T) for any property that depends on constraints imposed by T; other properties are stated in terms of the base type of T.

3   The first group of attributes yield characteristics of the subtype T. This group includes the attribute BASE (see 3.3.2), the attributes FIRST and LAST (see 3.5), the representation attribute SIZE (see 13.7.2), and the attribute WIDTH defined as follows:

4   T'WIDTH   Yields the maximum image length over all values of the subtype T (the *image* is the sequence of characters returned by the attribute IMAGE, see below). Yields zero for a null range. The value of this attribute is of the type *universal_integer.*

5   All attributes of the second group are functions with a single parameter. The corresponding actual parameter is indicated below by X.

6   T'POS   This attribute is a function. The parameter X must be a value of the base type of T. The result type is the type *universal_integer.* The result is the position number of the value of the parameter.

7   T'VAL   This attribute is a special function with a single parameter which can be of any integer type. The result type is the base type of T. The result is the value whose position number is the *universal_integer* value corresponding to X. The exception CONSTRAINT_ERROR is raised if the *universal_integer* value corresponding to X is not in the range T'POS(T'BASE'FIRST) .. T'POS(T'BASE'LAST).

8   T'SUCC   This attribute is a function. The parameter X must be a value of the base type of T. The result type is the base type of T. The result is the value whose position number is one greater than that of X. The exception CONSTRAINT_ERROR is raised if X equals T'BASE'LAST.

9   T´PRED   This attribute is a function. The parameter X must be a value of the base type of T. The result type is the base type of T. The result is the value whose position number is one less than that of X. The exception CONSTRAINT_ERROR is raised if X equals T´BASE´FIRST.

10   T´IMAGE   This attribute is a function. The parameter X must be a value of the base type of T. The result type is the predefined type STRING. The result is the *image* of the value of X, that is, a sequence of characters representing the value in display form. The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent, or trailing spaces; but with a single leading character that is either a minus sign or a space. The lower bound of the image is one. [13]

11   The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. The image of a character C, other than a graphic character, is implementation-defined; the only requirement is that the image must be such that C equals CHARACTER´VALUE(CHARACTER´IMAGE(C)).[14]

In DEC Ada, the image of a character C that is not a graphic character is defined as a string of two or three upper case letters without enclosing quotation marks or apostrophes. The upper case letters used are those shown in italics as the literals of the predefined type CHARACTER in Annex C (package STANDARD).

12   T´VALUE   This attribute is a function. The parameter X must be a value of the predefined type STRING. The result type is the base type of T. Any leading and any trailing spaces of the sequence of characters that corresponds to the parameter are ignored.

---

[13] See also Appendix G, AI-00234.
[14] See also Appendix G, AI-00239.

13   For an enumeration type, if the sequence of characters
      has the syntax of an enumeration literal and if this
      literal exists for the base type of T, the result is the
      corresponding enumeration value. For an integer type, if
      the sequence of characters has the syntax of an integer
      literal, with an optional single leading character that
      is a plus or minus sign, and if there is a corresponding
      value in the base type of T, the result is this value. In
      any other case, the exception CONSTRAINT_ERROR is
      raised.

14   In addition, the attributes A′SIZE and A′ADDRESS are defined for an object
      A of a discrete type (see 13.7.2).

15   Besides the basic operations, the operations of a discrete type include the
      predefined relational operators. For enumeration types, operations include
      enumeration literals. For boolean types, operations include the predefined
      unary logical negation operator **not**, and the predefined logical operators. For
      integer types, operations include the predefined *arithmetic* operators: these are
      the binary and unary adding operators – and +, all multiplying operators, the
      unary operator **abs**, and the exponentiating operator.

16   The operations of a subtype are the corresponding operations of its base type
      except for the following: assignment, membership tests, qualification, explicit
      type conversions, and the attributes of the first group; the effect of each of
      these operations depends on the subtype (assignments, membership tests,
      qualifications, and conversions involve a subtype check; attributes of the first
      group yield a characteristic of the subtype).

Notes:

17   For a subtype of a discrete type, the results delivered by the attributes SUCC,
      PRED, VAL, and VALUE need not belong to the subtype; similarly, the actual
      parameters of the attributes POS, SUCC, PRED, and IMAGE need not belong
      to the subtype. The following relations are satisfied (in the absence of an
      exception) by these attributes:

```
T'POS(T'SUCC(X)) = T'POS(X) + 1
T'POS(T'PRED(X)) = T'POS(X) - 1

T'VAL(T'POS(X))  = X
T'POS(T'VAL(N))  = N
```

```
--  For the types and subtypes declared
--  in section 3.5.1 we have:

--  COLOR'FIRST   = WHITE,   COLOR'LAST   = BLACK
--  RAINBOW'FIRST = RED,     RAINBOW'LAST = BLUE

--  COLOR'SUCC(BLUE) = RAINBOW'SUCC(BLUE) = BROWN
--  COLOR'POS(BLUE)  = RAINBOW'POS(BLUE)  = 4
--  COLOR'VAL(0)     = RAINBOW'VAL(0)     = WHITE
```

19   **References:** abs operator 4.5 4.5.6, assignment 5.2, attribute 4.1.4, base type 3.3, basic operation 3.3.3, binary adding operator 4.5 4.5.3, boolean type 3.5.3, bound of a range 3.5, character literal 2.5, constraint 3.3, constraint_error exception 11.1, conversion 4.6, discrete type 3.5, enumeration literal 3.5.1, exponentiating operator 4.5 4.5.6, function 6.5, graphic character 2.1, identifier 2.3, integer type 3.5.4, logical operator 4.5 4.5.1, membership test 4.5 4.5.2, multiplying operator 4.5 4.5.5, not operator 4.5 4.5.6, numeric literal 2.4, numeric type 3.5, object 3.2, operation 3.3, position number 3.5, predefined operator 4.5, predefined type C, qualified expression 4.7, relational operator 4.5 4.5.2, short-circuit control form 4.5 4.5.1, string type 3.6.3, subtype 3.3, type 3.3, unary adding operator 4.5 4.5.4, universal_integer type 3.5.4, universal type 4.10

character 2.1, character type 3.5.2, standard predefined package 8.6 C

3.5.6 Real Types

1   Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds for fixed point types.

```
2   real_type_definition ::=
        floating_point_constraint | fixed_point_constraint
```

3   A set of numbers called *model numbers* is associated with each real type. Error bounds on the predefined operations are given in terms of the model numbers. An implementation of the type must include at least these model numbers and represent them exactly.

4   An implementation-dependent set of numbers, called the *safe numbers*, is also associated with each real type. The set of safe numbers of a real type must include at least the set of model numbers of the type. The range of safe numbers is allowed to be larger than the range of model numbers, but error bounds on the predefined operations for safe numbers are given by the same rules as for model numbers. Safe numbers therefore provide guaranteed error bounds for operations on an implementation-dependent range of numbers; in contrast, the range of model numbers depends only on the real type definition and is therefore independent of the implementation.

5   Real literals are the literals of an anonymous predefined real type that is called *universal_real* in this reference manual. Other real types have no literals. However, for each real type, there exists an implicit conversion that converts a *universal_real* value into a value of the real type. The conditions under which these implicit conversions are invoked are described in section 4.6. If the *universal_real* value is a safe number, the implicit conversion delivers the corresponding value; if it belongs to the range of safe numbers but is not a safe number, then the converted value can be any value within the range defined by the safe numbers next above and below the *universal_real* value.

6   The execution of an operation that yields a value of a real type may raise the exception NUMERIC_ERROR, as explained in section 4.5.7, if it cannot deliver a correct result (that is, if the value corresponding to one of the possible mathematical results does not belong to the range of safe numbers); in particular, this exception can be raised by an implicit conversion. However, an implementation is not required to raise the exception NUMERIC_ERROR if the operation is part of a larger expression whose result can be computed correctly (see 11.6). [15]

7   The elaboration of a real type definition includes the elaboration of the floating or fixed point constraint and creates a real type.

Note:

8   An algorithm written to rely only upon the minimum numerical properties guaranteed by the type definition for model numbers will be portable without further precautions.

9   **References:** conversion 4.6, elaboration 3.9, fixed point constraint 3.5.9, floating point constraint 3.5.7, literal 4.2, numeric_error exception 11.1, predefined operation 3.3.3, real literal 2.4, type 3.3, type definition 3.3.1, universal type 4.10

3.5.7 Floating Point Types

1   For floating point types, the error bound is specified as a relative precision by giving the required minimum number of significant decimal digits.

```
2       floating_point_constraint ::=
            floating_accuracy_definition [range_constraint]

        floating_accuracy_definition ::=
          digits static_simple_expression
```

---

[15] See also Appendix G, AI-00387.

3 The minimum number of significant decimal digits is specified by the value of the static simple expression of the floating accuracy definition. This value must belong to some integer type and must be positive (nonzero); it is denoted by D in the remainder of this section. If the floating point constraint is used as a real type definition and includes a range constraint, then each bound of the range must be defined by a static expression of some real type, but the two bounds need not have the same real type.

4 For a given *radix*, the following canonical form is defined for any floating point model number other than zero:

```
sign * mantissa * (radix ** exponent)
```

5 In this form: *sign* is either +1 or −1; *mantissa* is expressed in a number base given by *radix*; and *exponent* is an integer number (possibly negative) such that the integer part of mantissa is zero and the first digit of its fractional part is not a zero.

6 The specified number D is the minimum number of decimal digits required after the point in the decimal mantissa (that is, if *radix* is ten). The value of D in turn determines a corresponding number B that is the minimum number of binary digits required after the point in the binary mantissa (that is, if *radix* is two). The number B associated with D is the smallest value such that the relative precision of the binary form is no less than that specified for the decimal form. (The number B is the integer next above $(D*\log(10)/\log(2)) + 1$.) [16]

7 The model numbers defined by a floating accuracy definition comprise zero and all numbers whose binary canonical form has exactly B digits after the point in the mantissa and an exponent in the range $-4*B$ .. $+4*B$. The guaranteed minimum accuracy of operations of a floating point type is defined in terms of the model numbers of the floating point constraint that forms the corresponding real type definition (see 4.5.7).

8 The predefined floating point types include the type FLOAT. An implementation may also have predefined types such as SHORT_FLOAT and LONG_FLOAT, which have (substantially) less and more accuracy, respectively, than FLOAT. The base type of each predefined floating point type is the type itself. The model numbers of each predefined floating point type are defined in terms of the number D of decimal digits returned by the attribute DIGITS (see 3.5.8).

---

[16] See also Appendix G, AI-00205.

DEC Ada provides the following floating point types in the package STANDARD. Additional floating point types are declared in the package SYSTEM (see 13.7b.1 and 13.7b.2).

Predefined type	DEC Ada systems on which it applies	Default representation[1]	Size (bits)	Digits of precision
FLOAT	OpenVMS	F_floating	32	6
	Digital UNIX	IEEE single float	32	6
LONG_FLOAT	OpenVMS	G_floating	64	15
	Digital UNIX	IEEE double float	64	15
LONG_LONG_FLOAT	OpenVMS VAX	H_floating	128	33
	OpenVMS Alpha	G_floating	64	15
	Digital UNIX	IEEE double float	64	15

[1] On some systems, the pragmas FLOAT_REPRESENTATION and LONG_FLOAT can be used to change the representation (see 3.5.7a and 3.5.7b).

The predefined attributes that yield the characteristics of each floating point type are described in section 3.5.8. Values of these attributes for the DEC Ada floating point data representations are listed in Appendix F. The DEC Ada run-time reference manuals also give information on the internal representation of the DEC Ada floating point types.

9 For each predefined floating point type (consequently also for each type derived therefrom), a set of safe numbers is defined as follows. The safe numbers have the same number B of mantissa digits as the model numbers of the type and have an exponent in the range –E .. +E where E is implementation-defined and at least equal to the 4*B of model numbers. (Consequently, the safe numbers include the model numbers.) The rules defining the accuracy of operations with model and safe numbers are given in section 4.5.7. The safe numbers of a subtype are those of its base type. [17]

10 A floating point type declaration of one of the two forms (that is, with or without the optional range constraint indicated by the square brackets): [18]

        type T **is digits** D [**range** L .. R];

11 is, by definition, equivalent to the following declarations:

        type *floating_point_type* **is new** predefined_floating_point_type;
        subtype T **is** *floating_point_type* **digits** D
            [**range** *floating_point_type*(L) .. *floating_point_type*(R)];

---

[17] See also Appendix G, AI-00217 and AI-00314.
[18] See also Appendix G, AI-00023.

12   where *floating_point_type* is an anonymous type, and where the predefined
      floating point type is implicitly selected by the implementation so that its
      model numbers include the model numbers defined by D; furthermore, if a
      range L .. R is supplied, then both L and R must belong to the range of safe
      numbers. The floating point declaration is illegal if none of the predefined
      floating point types satisfies these requirements, excepting *universal_real.*
      The maximum number of digits that can be specified in a floating accuracy
      definition is given by the system-dependent named number SYSTEM.MAX_
      DIGITS (see 13.7.1).

      The predefined attributes that yield the safe number characteristics of each
      floating point type are described in section 3.5.8. Values of these attributes for
      the DEC Ada floating point representations are listed in Appendix F.

13   The elaboration of a floating point type declaration consists of the elaboration
      of the equivalent type and subtype declarations.

14   If a floating point constraint follows a type mark in a subtype indication, the
      type mark must denote a floating point type or subtype. The floating point
      constraint is *compatible* with the type mark only if the number D specified in
      the floating accuracy definition is not greater than the corresponding number D
      for the type or subtype denoted by the type mark. Furthermore, if the floating
      point constraint includes a range constraint, the floating point constraint is
      compatible with the type mark only if the range constraint is, itself, compatible
      with the type mark.

15   The elaboration of such a subtype indication includes the elaboration of the
      range constraint, if there is one; it creates a floating point subtype whose
      model numbers are defined by the corresponding floating accuracy definition.
      A value of a floating point type belongs to a floating point subtype if and only
      if it belongs to the range defined by the subtype.

16   The same arithmetic operators are predefined for all floating point types
      (see 4.5).

      Notes:

17   A range constraint is allowed in a floating point subtype indication, either
      directly after the type mark, or as part of a floating point constraint. In either
      case the bounds of the range must belong to the base type of the type mark
      (see 3.5). The imposition of a floating point constraint on a type mark in a
      subtype indication cannot reduce the allowed range of values unless it includes
      a range constraint (the range of model numbers that correspond to the specified
      number of digits can be smaller than the range of numbers of the type mark).

A value that belongs to a floating point subtype need not be a model number of the subtype. [19]

18 **Examples:**

```
type COEFFICIENT is digits 10 range -1.0 .. 1.0;

type REAL is digits 8;
type MASS is digits 7 range 0.0 .. 1.0E35;

subtype SHORT_COEFF is COEFFICIENT digits 5;  -- a subtype with
                                              -- less accuracy

subtype PROBABILITY is REAL range 0.0 .. 1.0; -- a subtype with
                                              -- a smaller range
```

Notes on the examples:

19 The implemented accuracy for COEFFICIENT is that of a predefined type having at least 10 digits of precision. Consequently the specification of 5 digits of precision for the subtype SHORT_COEFF is allowed. The largest model number for the type MASS is approximately 1.27E30 and hence less than the specified upper bound (1.0E35). Consequently the declaration of this type is legal only if this upper bound is in the range of the safe numbers of a predefined floating point type having at least 7 digits of precision.

20 **References:** anonymous type 3.3.1, arithmetic operator 3.5.5 4.5, based literal 2.4.2, belong to a subtype 3.3, bound of a range 3.5, compatible 3.3.2, derived type 3.4, digit 2.1, elaboration 3.1 3.9, error bound 3.5.6, exponent 2.4.1 integer type 3.5.4, model number 3.5.6, operation 3.3, predefined operator 4.5, predefined type C, range constraint 3.5, real type 3.5.6, real type definition 3.5.6, safe number 3.5.6, simple expression 4.4, static expression 4.9, subtype declaration 3.3.2, subtype indication 3.3.2, subtype 3.3, type 3.3, type declaration 3.3.1, type mark 3.3.2

float_representation pragma 3.5.7a, long_float pragma 3.5.7b

3.5.7a Pragma Float_Representation

On OpenVMS and Digital UNIX systems, DEC Ada provides the pragma FLOAT_REPRESENTATION to allow control over the internal representation chosen for the predefined floating point types declared in the packages STANDARD and SYSTEM. The form of this pragma is as follows:

```
pragma FLOAT_REPRESENTATION(VAX_FLOAT |  -- on OpenVMS systems
                            IEEE_FLOAT); -- on Alpha systems
```

---

[19] See also Appendix G, AI-00375.

This pragma is only allowed at the start of a compilation before the first compilation unit (if any). It controls the representation of floating point types as follows:

- If the value VAX_FLOAT is specified, then floating point types are represented with the VAX hardware types F_floating, D_floating, G_floating, and H_floating. The type H_floating is available on OpenVMS VAX systems only.

  If the value is VAX_FLOAT, the default representation of the predefined type STANDARD.LONG_FLOAT is G_floating. The representation can be changed with the pragma LONG_FLOAT (see 3.5.7b).

- If the value IEEE_FLOAT is specified, then floating point types are represented with the IEEE single and double floating types.

On OpenVMS VAX systems, the only value allowed is VAX_FLOAT.

On OpenVMS Alpha systems, the values allowed are VAX_FLOAT and IEEE_FLOAT. Note that the predefined library is compiled with VAX_FLOAT, and VAX_FLOAT is required for interface to system routine calls. An explicit type conversion may be necessary in order to use the predefined library.

On Digital UNIX systems, the only value allowed is IEEE_FLOAT.

The following table summarizes by platform the allowable and default values (chosen whenever a program library is reinitialized):

Platform	Allowable Values	Default
OpenVMS VAX	VAX_FLOAT	VAX_FLOAT
OpenVMS Alpha	VAX_FLOAT IEEE_FLOAT	VAX_FLOAT
Digital UNIX	IEEE_FLOAT	IEEE_FLOAT

Use of the pragma FLOAT_REPRESENTATION is interpreted as an implicit recompilation of the predefined STANDARD environment. Therefore, if the value of the pragma FLOAT_REPRESENTATION changes, all units that depend on the value must be recompiled.

See section 3.5.7 for more information about the floating point types predefined by DEC Ada.

Notes:

All representation choices also depend on the range of values to be represented. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* or *DEC Ada Run-Time Reference Manual for DEC OSF/1 Systems* for more information.

References: allow 1.6, compilation unit 10.1, d_floating representation 3.5.7, f_floating representation 3.5.7, floating point type declaration 3.5.7, g_floating representation 3.5.7, h_floating representation 3.5.7, ieee single floating representation 3.5.7, ieee double floating representation 3.5.7, long_float type 3.5.7, order of compilation 10.3, pragma 2.8, program library 10.1, standard package 8.6 Appendix C, system package Appendix F

3.5.7b Pragma Long_Float (OpenVMS Systems Only)

On OpenVMS systems, DEC Ada provides the pragma LONG_FLOAT to allow control over the internal representation chosen for the predefined type LONG_FLOAT and for floating point type declarations with **digits** specified in the range 7 .. 15. The form of this pragma is as follows:

```
pragma LONG_FLOAT(D_FLOAT | G_FLOAT);
```

This pragma is only allowed at the start of a compilation before the first compilation unit (if any). For this pragma to have an effect, the pragma FLOAT_REPRESENTATION must also have a value of VAX_FLOAT (see 3.5.7a).

If the value D_FLOAT is specified and the range is adequate, then a D_floating representation is used for the predefined type LONG_FLOAT and for floating point types declared with **digits** in the range 7 .. 9. Similarly, if the value G_FLOAT is specified and the range is adequate, then a G_floating representation is used for the predefined type LONG_FLOAT and for floating point types declared with **digits** in the range 7 .. 15. F_floating and H_floating representations are used for floating point types with **digits** in other ranges as follows:

Pragma argument	Digits specified	Representation
D_FLOAT	1 .. 6	F_floating
	7 .. 9	D_floating
	10 .. 15	G_floating[1]
	10 .. 33	H_floating[2]

[1]On Alpha systems.

[2]On VAX systems.

Pragma argument	Digits specified	Representation
G_FLOAT	1 .. 6	F_floating
	7 .. 15	G_floating
	16 .. 33	H_floating[2]

---

[2] On VAX systems.

Use of the pragma LONG_FLOAT is interpreted as an implicit recompilation of the predefined STANDARD environment. Therefore, if the value of the pragma LONG_FLOAT changes, all units that depend on the value must be recompiled. Whenever a program library is reinitialized, the G_floating representation is established by default.

Notes:

When the value of the pragma FLOAT_REPRESENTATION is VAX_FLOAT (see 3.5.7a), the choice of floating point representation is not affected by the pragma LONG_FLOAT in the following cases:

Digits specified	Representation	DEC Ada systems on which it applies
1 .. 6	F_floating	All OpenVMS
10 .. 15	G_floating	OpenVMS Alpha
16 .. 33	H_floating	OpenVMS VAX

All representation choices also depend on the range of values to be represented. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

References: allow 1.6, compilation unit 10.1, d_float type 3.5.7, d_floating representation 3.5.7, f_float type 3.5.7, f_floating representation 3.5.7, floating point type declaration 3.5.7, float_representation pragma 3.5.7a, g_float type 3.5.7, g_floating representation 3.5.7, h_float type 3.5.7, h_floating representation 3.5.7, long_float type 3.5.7, order of compilation 10.3, pragma 2.8, program library 10.1, range 3.5, standard package 8.6 Appendix C

3.5.8 Operations of Floating Point Types

1   The basic operations of a floating point type include the operations involved in assignment, membership tests, qualification, the explicit conversion of values of other numeric types to the floating point type, and the implicit conversion of values of the type *universal_real* to the type.

2   In addition, for every floating point type or subtype T, the basic operations include the attributes listed below. In this presentation, T is referred to as being a subtype (the subtype T) for any property that depends on constraints imposed by T; other properties are stated in terms of the base type of T.

Appendix F gives values for these attributes for each of the DEC Ada floating point representations.

3   The first group of attributes yield characteristics of the subtype T. The attributes of this group are the attribute BASE (see 3.3.2), the attributes FIRST and LAST (see 3.5), the representation attribute SIZE (see 13.7.2), and the following attributes:

4   T'DIGITS   Yields the number of decimal digits in the decimal mantissa of model numbers of the subtype T. (This attribute yields the number D of section 3.5.7.) The value of this attribute is of the type *universal_integer.*

5   T'MANTISSA   Yields the number of binary digits in the binary mantissa of model numbers of the subtype T. (This attribute yields the number B of section 3.5.7.) The value of this attribute is of the type *universal_integer.*

6   T'EPSILON   Yields the absolute value of the difference between the model number 1.0 and the next model number above, for the subtype T. The value of this attribute is of the type *universal_real.*

7   T'EMAX   Yields the largest exponent value in the binary canonical form of model numbers of the subtype T. (This attribute yields the product 4*B of section 3.5.7.) The value of this attribute is of the type *universal_integer.*

8   T'SMALL   Yields the smallest positive (nonzero) model number of the subtype T. The value of this attribute is of the type *universal_real.*

9   T'LARGE   Yields the largest positive model number of the subtype T. The value of this attribute is of the type *universal_real.*

10   The attributes of the second group include the following attributes which yield characteristics of the safe numbers:

11   T'SAFE_EMAX   Yields the largest exponent value in the binary canonical form of safe numbers of the base type of T. (This attribute yields the number E of section 3.5.7.) The value of this attribute is of the type *universal_integer.*

12   T'SAFE_SMALL Yields the smallest positive (nonzero) safe number of the base type of T. The value of this attribute is of the type *universal_real.*

13   T'SAFE_LARGE Yields the largest positive safe number of the base type of T. The value of this attribute is of the type *universal_ real.*

14   In addition, the attributes A'SIZE and A'ADDRESS are defined for an object A of a floating point type (see 13.7.2). Finally, for each floating point type there are machine-dependent attributes that are not related to model numbers and safe numbers. They correspond to the attribute designators MACHINE_ RADIX, MACHINE_MANTISSA, MACHINE_EMAX, MACHINE_EMIN, MACHINE_ROUNDS, and MACHINE_OVERFLOWS (see 13.7.3).

Appendix F gives values for all of the machine-dependent attributes for each of the DEC Ada floating point representations.

15   Besides the basic operations, the operations of a floating point type include the relational operators, and the following predefined arithmetic operators: the binary and unary adding operators – and +, the multiplying operators * and /, the unary operator **abs**, and the exponentiating operator.

16   The operations of a subtype are the corresponding operations of the type except for the following: assignment, membership tests, qualification, explicit conversion, and the attributes of the first group; the effects of these operations are redefined in terms of the subtype. [20]

Notes:

17   The attributes EMAX, SMALL, LARGE, and EPSILON are provided for convenience. They are all related to MANTISSA by the following formulas:

```
T'EMAX    = 4*T'MANTISSA
T'EPSILON = 2.0**(1 - T'MANTISSA)
T'SMALL   = 2.0**(-T'EMAX - 1)
T'LARGE   = 2.0**T'EMAX * (1.0 - 2.0**(-T'MANTISSA))
```

18   The attribute MANTISSA, giving the number of binary digits in the mantissa, is itself related to DIGITS. The following relations hold between the characteristics of the model numbers and those of the safe numbers:

```
T'BASE'EMAX  <= T'SAFE_EMAX
T'BASE'SMALL >= T'SAFE_SMALL
T'BASE'LARGE <= T'SAFE_LARGE
```

---

[20] See also Appendix G, AI-00407.

19   The attributes T'FIRST and T'LAST need not yield model or safe numbers. If a certain number of digits is specified in the declaration of a type or subtype T, the attribute T'DIGITS yields this number.

20   **References:** abs operator 4.5 4.5.6, arithmetic operator 3.5.5 4.5, assignment 5.2, attribute 4.1.4, base type 3.3, basic operation 3.3.3, binary adding operator 4.5 4.5.3, bound of a range 3.5, constraint 3.3, conversion 4.6, digit 2.1, exponentiating operator 4.5 4.5.6, floating point type 3.5.7, membership test 4.5 4.5.2, model number 3.5.6, multiplying operator 4.5 4.5.5, numeric type 3.5, object 3.2, operation 3.3, predefined operator 4.5, qualified expression 4.7, relational operator 4.5 4.5.2, safe number 3.5.6, subtype 3.3, type 3.3, unary adding operator 4.5 4.5.4, universal type 4.10, universal_ integer type 3.5.4, universal_real type 3.5.6

floating point representation 3.5.7

3.5.9 Fixed Point Types

1   For fixed point types, the error bound is specified as an absolute value, called the *delta* of the fixed point type.

2
```
fixed_point_constraint ::=
    fixed_accuracy_definition [range_constraint]

fixed_accuracy_definition ::=  delta static_simple_expression
```

3   The delta is specified by the value of the static simple expression of the fixed accuracy definition. This value must belong to some real type and must be positive (nonzero). If the fixed point constraint is used as a real type definition, then it must include a range constraint; each bound of the specified range must be defined by a static expression of some real type but the two bounds need not have the same real type. If the fixed point constraint is used in a subtype indication, the range constraint is optional.

4   A canonical form is defined for any fixed point model number other than zero. In this form: *sign* is either +1 or −1; *mantissa* is a positive (nonzero) integer; and any model number is a multiple of a certain positive real number called *small*, as follows:

        *sign * mantissa * small*

5   For the model numbers defined by a fixed point constraint, the number *small* is chosen as the largest power of two that is not greater than the delta of the fixed accuracy definition. Alternatively, it is possible to specify the value of *small* by a length clause (see 13.2), in which case model numbers are multiples of the specified value. The guaranteed minimum accuracy of operations of a fixed point type is defined in terms of the model numbers of the fixed point constraint that forms the corresponding real type definition (see 4.5.7).

6   For a fixed point constraint that includes a range constraint, the model numbers comprise zero and all multiples of *small* whose *mantissa* can be expressed using exactly B binary digits, where the value of B is chosen as the smallest integer number for which each bound of the specified range is either a model number or lies at most *small* distant from a model number. For a fixed point constraint that does not include a range constraint (this is only allowed after a type mark, in a subtype indication), the model numbers are defined by the delta of the fixed accuracy definition and by the range of the subtype denoted by the type mark. [21]

7   An implementation must have at least one anonymous predefined fixed point type. The base type of each such fixed point type is the type itself. The model numbers of each predefined fixed point type comprise zero and all numbers for which *mantissa* (in the canonical form) has the number of binary digits returned by the attribute MANTISSA, and for which the number *small* has the value returned by the attribute SMALL.

To implement fixed point numbers, DEC Ada uses a set of anonymous predefined fixed point types of the form:

```
type fixed_point_type is delta S range -L .. U;
```

where S is a value in the range 2.0^{-62} .. 2.0^{31}, L = 2.0^{31}*S, and U = $(2.0^{31} - 1)$*S. Each fixed point type has a size determined by its delta and range, rounded up to an 8-, 16-, or 32-bit boundary. The size may be changed by a representation clause (see 13.1).

8   A fixed point type declaration of the form: [22]

```
type T is delta D range L .. R;
```

9   is, by definition, equivalent to the following declarations: [23]

```
type fixed_point_type is new predefined_fixed_point_type;
subtype T is fixed_point_type
   range fixed_point_type(L) .. fixed_point_type(R);
```

10 In these declarations, *fixed_point_type* is an anonymous type, and the predefined fixed point type is implicitly selected by the implementation so that its model numbers include the model numbers defined by the fixed point constraint (that is, by D, L, and R, and possibly by a length clause specifying *small*). [24]

---

[21] See also Appendix G, AI-00143.
[22] See also Appendix G, AI-00023.
[23] See also Appendix G, AI-00144 and AI-00471.
[24] See also Appendix G, AI-00343.

11   The fixed point declaration is illegal if no predefined type satisfies these
      requirements. The safe numbers of a fixed point type are the model numbers
      of its base type. [25]

12   The elaboration of a fixed point type declaration consists of the elaboration of
      the equivalent type and subtype declarations.

13   If the fixed point constraint follows a type mark in a subtype indication,
      the type mark must denote a fixed point type or subtype. The fixed point
      constraint is *compatible* with the type mark only if the delta specified by the
      fixed accuracy definition is not smaller than the delta for the type or subtype
      denoted by the type mark. Furthermore, if the fixed point constraint includes
      a range constraint, the fixed point constraint is compatible with the type mark
      only if the range constraint is, itself, compatible with the type mark.

14   The elaboration of such a subtype indication includes the elaboration of the
      range constraint, if there is one; it creates a fixed point subtype whose model
      numbers are defined by the corresponding fixed point constraint and also by
      the length clause specifying small, if there is one. A value of a fixed point type
      belongs to a fixed point subtype if and only if it belongs to the range defined by
      the subtype. [26]

15   The same arithmetic operators are predefined for all fixed point types
      (see 4.5). Multiplication and division of fixed point values deliver results of
      an anonymous predefined fixed point type that is called *universal_fixed* in this
      reference manual; the accuracy of this type is arbitrarily fine. The values of
      this type must be converted explicitly to some numeric type.

      Notes:

16   If S is a subtype of a fixed point type or subtype T, then the set of model
      numbers of S is a subset of those of T. If a length clause has been given for
      T, then both S and T have the same value for *small.* Otherwise, since *small*
      is a power of two, the *small* of S is equal to the *small* of T multiplied by a
      nonnegative power of two. [27]

17   A range constraint is allowed in a fixed point subtype indication, either directly
      after the type mark, or as part of a fixed point constraint. In either case the
      bounds of the range must belong to the base type of the type mark (see 3.5).

---

[25] See also Appendix G, AI-00508.
[26] See also Appendix G, AI-00145 and AI-00146.
[27] See also Appendix G, AI-00146.

Examples:

```
type VOLT is delta 0.125 range 0.0 .. 255.0;
subtype ROUGH_VOLTAGE is VOLT delta 1.0;  --  same range as VOLT

--  A pure fraction which requires all the available space
--  in a word on a two's complement machine can be declared
--  as the type FRACTION:

DEL : constant := 1.0/2**(WORD_LENGTH - 1);
type FRACTION is delta DEL range
-1.0 .. 1.0 - DEL;[28]
```

19 **References:** anonymous type 3.3.1, arithmetic operator 3.5.5 4.5, base type 3.3, belong to a subtype 3.3, bound of a range 3.5, compatible 3.3.2, conversion 4.6, elaboration 3.9, error bound 3.5.6, length clause 13.2, model number 3.5.6, numeric type 3.5, operation 3.3, predefined operator 4.5, range constraint 3.5, real type 3.5.6, real type definition 3.5.6, safe number 3.5.6, simple expression 4.4, static expression 4.9, subtype 3.3, subtype declaration 3.3.2, subtype indication 3.3.2, type 3.3, type declaration 3.3.1, type mark 3.3.2

3.5.10 Operations of Fixed Point Types

1 The basic operations of a fixed point type include the operations involved in assignment, membership tests, qualification, the explicit conversion of values of other numeric types to the fixed point type, and the implicit conversion of values of the type *universal_real* to the type.

2 In addition, for every fixed point type or subtype T the basic operations include the attributes listed below. In this presentation T is referred to as being a subtype (the subtype T) for any property that depends on constraints imposed by T; other properties are stated in terms of the base type of T.

3 The first group of attributes yield characteristics of the subtype T. The attributes of this group are the attributes BASE (see 3.3.2), the attributes FIRST and LAST (see 3.5), the representation attribute SIZE (see 13.7.2) and the following attributes:

4   T'DELTA   Yields the value of the delta specified in the fixed accuracy definition for the subtype T. The value of this attribute is of the type *universal_real.*

5   T'MANTISSA   Yields the number of binary digits in the mantissa of model numbers of the subtype T. (This attribute yields the number B of section 3.5.9.) The value of this attribute is of the type *universal_integer.*

---

[28] See also Appendix G, AI-00147.

6	T'SMALL	Yields the smallest positive (nonzero) model number of the subtype T. The value of this attribute is of the type *universal_real.*
7	T'LARGE	Yields the largest positive model number of the subtype T. The value of this attribute is of the type *universal_ real.*
8	T'FORE	Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype T, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least two.) The value of this attribute is of the type *universal_integer.* [29]
9	T'AFT	Yields the number of decimal digits needed after the point to accommodate the precision of the subtype T, unless the delta of the subtype T is greater than 0.1, in which case the attribute yields the value one. (T'AFT is the smallest positive integer N for which $(10^{**}N)^*T'DELTA$ is greater than or equal to one.) The value of this attribute is of the type *universal_integer.*

10 The attributes of the second group include the following attributes which yield characteristics of the safe numbers:

11	T'SAFE_SMALL	Yields the smallest positive (nonzero) safe number of the base type of T. The value of this attribute is of the type *universal_real.*
12	T'SAFE_LARGE	Yields the largest positive safe number of the base type of T. The value of this attribute is of the type *universal_ real.*

13 In addition, the attributes A'SIZE and A'ADDRESS are defined for an object A of a fixed point type (see 13.7.2). Finally, for each fixed point type or subtype T, there are the machine-dependent attributes T'MACHINE_ROUNDS and T'MACHINE_OVERFLOWS (see 13.7.3).

14 Besides the basic operations, the operations of a fixed point type include the relational operators, and the following predefined arithmetic operators: the binary and unary adding operators – and +, the multiplying operators * and /, and the operator **abs.**

---

[29] See also Appendix G, AI-00179 and Section AI-00467.

15   The operations of a subtype are the corresponding operations of the type
      except for the following: assignment, membership tests, qualification, explicit
      conversion, and the attributes of the first group; the effects of these operations
      are redefined in terms of the subtype. [30]

      Notes:

16   The value of the attribute T′FORE depends only on the range of the subtype
      T. The value of the attribute T′AFT depends only on the value of T′DELTA.
      The following relations exist between attributes of a fixed point type:

```
T'LARGE      = (2**T'MANTISSA - 1) * T'SMALL
T'SAFE_LARGE = T'BASE'LARGE
T'SAFE_SMALL = T'BASE'SMALL
```

17   **References:** abs operator 4.5 4.5.6, arithmetic operator 3.5.5 4.5, assignment 5.2,
      base type 3.3, basic operation 3.3.3, binary adding operator 4.5 4.5.3, bound of a range
      3.5, conversion 4.6, delta 3.5.9, fixed point type 3.5.9, membership test 4.5 4.5.2, model
      number 3.5.6, multiplying operator 4.5 4.5.5, numeric type 3.5, object 3.2, operation
      3.3, qualified expression 4.7, relational operator 4.5 4.5.2, safe number 3.5.6, subtype
      3.3, unary adding operator 4.5 4.5.4, universal_integer type 3.5.4, universal_real type
      3.5.6

3.6 Array Types

1   An array object is a composite object consisting of components that have the
      same subtype. The name for a component of an array uses one or more index
      values belonging to specified discrete types. The value of an array object is a
      composite value consisting of the values of its components.

2
```
array_type_definition ::=
      unconstrained_array_definition
    | constrained_array_definition

unconstrained_array_definition ::=
   array(index_subtype_definition
            {, index_subtype_definition}) of
            component_subtype_indication

constrained_array_definition ::=
   array index_constraint of component_subtype_indication

index_subtype_definition ::= type_mark range <>

index_constraint ::=  (discrete_range {, discrete_range})

discrete_range ::= discrete_subtype_indication | range
```

---

[30] See also Appendix G, AI-00407.

3   An array object is characterized by the number of indices (the *dimensionality* of the array), the type and position of each index, the lower and upper bounds for each index, and the type and possible constraint of the components. The order of the indices is significant.

4   A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive; this range of values is called the *index range.*

5   An unconstrained array definition defines an array type. For each object that has the array type, the number of indices, the type and position of each index, and the subtype of the components are as in the type definition; the values of the lower and upper bounds for each index belong to the corresponding index subtype, except for null arrays as explained in section 3.6.1. The *index subtype* for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. The compound delimiter <> (called a *box*) of an index subtype definition stands for an undefined range (different objects of the type need not have the same bounds). The elaboration of an unconstrained array definition creates an array type; this elaboration includes that of the component subtype indication.

6   A constrained array definition defines both an array type and a subtype of this type:

7   • The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unconstrained array definition, in which the component subtype indication is that of the constrained array definition, and in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range.

8   • The array subtype is the subtype obtained by imposition of the index constraint on the array type.

9   If a constrained array definition is given for a type declaration, the simple name declared by this declaration denotes the array subtype.

10   The elaboration of a constrained array definition creates the corresponding array type and array subtype. For this elaboration, the index constraint and the component subtype indication are elaborated. The evaluation of each discrete range of the index constraint and the elaboration of the component subtype indication are performed in some order that is not defined by the language.

11   **Examples of type declarations with unconstrained array definitions:**

```
type VECTOR     is array(INTEGER  range <>) of REAL;
type MATRIX     is array(INTEGER  range <>,
                         INTEGER range <>) of REAL;
type BIT_VECTOR is array(INTEGER  range <>) of BOOLEAN;
type ROMAN      is array(POSITIVE range <>) of ROMAN_DIGIT;
```

12   **Examples of type declarations with constrained array definitions:**

```
type TABLE    is array(1 .. 10) of INTEGER;
type SCHEDULE is array(DAY) of BOOLEAN;
type LINE     is array(1 .. MAX_LINE_SIZE) of CHARACTER;
```

13   **Examples of object declarations with constrained array definitions:**

```
GRID : array(1 .. 80, 1 .. 100) of BOOLEAN;
MIX  : array(COLOR range RED .. GREEN) of BOOLEAN;
PAGE : array(1 .. 50) of LINE;  --  an array of arrays
```

Note:

14   For a one-dimensional array, the rule given means that a type declaration with
a constrained array definition such as

```
type T is array(POSITIVE range MIN .. MAX) of COMPONENT;
```

15   is equivalent (in the absence of an incorrect order dependence) to the succession
of declarations

```
subtype index_subtype is POSITIVE range MIN .. MAX;
type array_type is array(index_subtype range <>) of COMPONENT;
subtype T is array_type(index_subtype);
```

16   where *index_subtype* and *array_type* are both anonymous. Consequently, T is
the name of a subtype and all objects declared with this type mark are arrays
that have the same bounds. Similar transformations apply to multidimensional
arrays.

17   A similar transformation applies to an object whose declaration includes a
constrained array definition. A consequence of this is that no two such objects
have the same type.

18   **References:** anonymous type 3.3.1, bound of a range 3.5, component 3.3, constraint
3.3, discrete type 3.5, elaboration 3.1 3.9, in some order 1.6, name 4.1, object 3.2,
range 3.5, subtype 3.3, subtype indication 3.3.2, type 3.3, type declaration 3.3.1, type
definition 3.3.1, type mark 3.3.2

3.6.1 Index Constraints and Discrete Ranges

1 An index constraint determines the range of possible values for every index of an array type, and thereby the corresponding array bounds.

2 For a discrete range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if each bound is either a numeric literal, a named number, or an attribute, and the type of both bounds (prior to the implicit conversion) is the type *universal_integer.* Otherwise, both bounds must be of the same discrete type, other than *universal_integer*; this type must be determinable independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration rule (see 5.5) or in the declaration of a family of entries (see 9.5). [31]

3 If an index constraint follows a type mark in a subtype indication, then the type or subtype denoted by the type mark must not already impose an index constraint. The type mark must denote either an unconstrained array type or an access type whose designated type is such an array type. In either case, the index constraint must provide a discrete range for each index of the array type and the type of each discrete range must be the same as that of the corresponding index.

4 An index constraint is *compatible* with the type denoted by the type mark if and only if the constraint defined by each discrete range is compatible with the corresponding index subtype. If any of the discrete ranges defines a null range, any array thus constrained is a *null array*, having no components. An array value *satisfies* an index constraint if at each index position the array value and the index constraint have the same index bounds. (Note, however, that assignment and certain other operations on arrays involve an implicit subtype conversion.) [32]

5 The bounds of each array object are determined as follows:

6 • For a variable declared by an object declaration, the subtype indication of the corresponding object declaration must define a constrained array subtype (and, thereby, the bounds). The same requirement exists for the subtype indication of a component declaration, if the type of the record component is an array type; and for the component subtype indication of an array type definition, if the type of the array components is itself an array type.

---

[31] See also Appendix G, AI-00148.
[32] See also Appendix G, AI-00282.

7 • For a constant declared by an object declaration, the bounds of the constant are defined by the initial value if the subtype of the constant is unconstrained; they are otherwise defined by this subtype (in the latter case, the initial value is the result of an implicit subtype conversion). The same rule applies to a generic formal parameter of mode **in.**

8 • For an array object designated by an access value, the bounds must be defined by the allocator that creates the array object. (The allocated object is constrained with the corresponding values of the bounds.)

9 • For a formal parameter of a subprogram or entry, the bounds are obtained from the corresponding actual parameter. (The formal parameter is constrained with the corresponding values of the bounds.)

10 • For a renaming declaration and for a generic formal parameter of mode **in out**, the bounds are those of the renamed object or of the corresponding generic actual parameter.

11 For the elaboration of an index constraint, the discrete ranges are evaluated in some order that is not defined by the language.

12 **Examples of array declarations including an index constraint:**

```
BOARD     : MATRIX(1 .. 8,  1 .. 8);  --  see 3.6
RECTANGLE : MATRIX(1 .. 20, 1 .. 30);
INVERSE   : MATRIX(1 .. N,  1 .. N);  --  N need not be static

FILTER    : BIT_VECTOR(0 .. 31);
```

13 **Example of array declaration with a constrained array subtype:**

```
MY_SCHEDULE : SCHEDULE;  --  all arrays of type SCHEDULE
                         --  have the same bounds
```

14 **Example of record type with a component that is an array:**

```
type VAR_LINE(LENGTH : INTEGER) is
   record
      IMAGE : STRING(1 .. LENGTH);
   end record;

NULL_LINE : VAR_LINE(0);  --  NULL_LINE.IMAGE is a null array
```

Notes:

15 The elaboration of a subtype indication consisting of a type mark followed by an index constraint checks the compatibility of the index constraint with the type mark (see 3.3.2).

16 All components of an array have the same subtype. In particular, for an array of components that are one-dimensional arrays, this means that all components have the same bounds and hence the same length.

[17] **References:** access type 3.8, access type definition 3.8, access value 3.8, actual parameter 6.4.1, allocator 4.8, array bound 3.6, array component 3.6, array type 3.6, array type definition 3.6, bound of a range 3.5, compatible 3.3.2, component declaration 3.7, constant 3.2.1, constrained array definition 3.6, constrained array subtype 3.6, conversion 4.6, designate 3.8, designated type 3.8, discrete range 3.6, entry 9.5, entry family declaration 9.5, expression 4.4, formal parameter 6.1, function 6.5, generic actual parameter 12.3, generic formal parameter 12.1 12.3, generic parameter 12.1, index 3.6, index constraint 3.6.1, index subtype 3.6, initial value 3.2.1, integer literal 2.4, integer type 3.5.4, iteration rule 5.5, mode 12.1.1, name 4.1, null range 3.5, object 3.2, object declaration 3.2.1, predefined type C, range 3.5, record component 3.7, renaming declaration 8.5, result subtype 6.1, satisfy 3.3, subprogram 6, subtype conversion 4.6, subtype indication 3.3.2, type mark 3.3.2, unconstrained array type 3.6, unconstrained subtype 3.3, universal type 4.10, universal_integer type 3.5.4, variable 3.2.1

3.6.2 Operations of Array Types

[1] The basic operations of an array type include the operations involved in assignment and aggregates (unless the array type is limited), membership tests, indexed components, qualification, and explicit conversion; for one-dimensional arrays the basic operations also include the operations involved in slices, and also string literals if the component type is a character type.

[2] If A is an array object, an array value, or a constrained array subtype, the basic operations also include the attributes listed below. These attributes are not allowed for an unconstrained array type. The argument N used in the attribute designators for the N-th dimension of an array must be a static expression of type *universal_integer*. The value of N must be positive (nonzero) and no greater than the dimensionality of the array.

[3]   A′FIRST   Yields the lower bound of the first index range. The value of this attribute has the same type as this lower bound.

[4]   A′FIRST(N)   Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound.

[5]   A′LAST   Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound.

[6]   A′LAST(N)   Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound.

[7]   A′RANGE   Yields the first index range, that is, the range A′FIRST .. A′LAST.

8   A'RANGE(N)   Yields the N-th index range, that is, the range
                                 A'FIRST(N) .. A'LAST(N).

9   A'LENGTH   Yields the number of values of the first index range
                                 (zero for a null range). The value of this attribute is
                                 of the type *universal_integer.*

10   A'LENGTH(N)   Yields the number of values of the N-th index range
                                 (zero for a null range). The value of this attribute is
                                 of the type *universal_integer.*

11   In addition, the attribute T'BASE is defined for an array type or subtype T
     (see 3.3.3); the attribute T'SIZE is defined for an array type or subtype T, and
     the attributes A'SIZE and A'ADDRESS are defined for an array object A
     (see 13.7.2).

12   Besides the basic operations, the operations of an array type include the
     predefined comparison for equality and inequality, unless the array type
     is limited. For one-dimensional arrays, the operations include catenation,
     unless the array type is limited; if the component type is a discrete type, the
     operations also include all predefined relational operators; if the component
     type is a boolean type, then the operations also include the unary logical
     negation operator **not**, and the logical operators.

13   **Examples using arrays declared in the examples of section 3.6.1:**

```
--   FILTER'FIRST      =     0
--   FILTER'LAST       =    31
--   FILTER'LENGTH     =    32
--   RECTANGLE'LAST(1) =    20
--   RECTANGLE'LAST(2) =    30
```

Notes:

14   The attributes A'FIRST and A'FIRST(1) yield the same value. A similar
     relation exists for the attributes A'LAST, A'RANGE, and A'LENGTH. The
     following relations are satisfied (except for a null array) by the above attributes
     if the index type is an integer type:

```
A'LENGTH    = A'LAST    - A'FIRST    + 1
A'LENGTH(N) = A'LAST(N) - A'FIRST(N) + 1
```

15   An array type is limited if its component type is limited (see 7.4.4).

16   **References:** aggregate 4.3, array type 3.6, assignment 5.2, attribute 4.1.4, basic
     operation 3.3.3, bound of a range 3.5, catenation operator 4.5 4.5.3, character type
     3.5.2, constrained array subtype 3.6, conversion 4.6, designator 6.1, dimension 3.6,
     index 3.6, indexed component 4.1.1, limited type 7.4.4, logical operator 4.5 4.5.1,
     membership test 4.5 4.5.2, not operator 4.5 4.5.6, null range 3.5, object 3.2, operation
     3.3, predefined operator 4.5, qualified expression 4.7, relational operator 4.5 4.5.2, slice

4.1.2, static expression 4.9, string literal 2.6, subcomponent 3.3, type 3.3, unconstrained array type 3.6, universal type 4.10, universal_integer type 3.5.4

3.6.3 The Type String

1   The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE:

```
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
type STRING is array(POSITIVE range <>) of CHARACTER;
```

2   **Examples:**
```
STARS       : STRING(1 .. 120) := (1 .. 120 => '*' );
QUESTION    : constant STRING  := "HOW MANY CHARACTERS?";
--  QUESTION'FIRST = 1,
--  QUESTION'LAST = 20 (the number of characters)

ASK_TWICE  : constant STRING  := QUESTION & QUESTION;
NINETY_SIX : constant ROMAN   := "XCVI";        -- see 3.6
```

   Notes:

3   String literals (see 2.6 and 4.2) are basic operations applicable to the type STRING and to any other one-dimensional array type whose component type is a character type. The catenation operator is a predefined operator for the type STRING and for one-dimensional array types; it is represented as &. The relational operators <, <=,>, and >= are defined for values of these types, and correspond to lexicographic order (see 4.5.2).

4   **References:** aggregate 4.3, array 3.6, catenation operator 4.5 4.5.3, character type 3.5.2, component type (of an array) 3.6, dimension 3.6, index 3.6, lexicographic order 4.5.2, positional aggregate 4.3, predefined operator 4.5, predefined type C, relational operator 4.5 4.5.2, string literal 2.6, subtype 3.3, type 3.3

3.7 Record Types

1   A record object is a composite object consisting of named components. The value of a record object is a composite value consisting of the values of its components.

2   ```
     record_type_definition ::=
         record
             component_list
         end record

     component_list ::=
             component_declaration {component_declaration}
         | {component_declaration} variant_part
         | null;
     ```

```
component_declaration ::=
   identifier_list : component_subtype_definition
                       [:= expression];

component_subtype_definition ::=  subtype_indication
```

3   Each component declaration declares a component of the record type. Besides
    components declared by component declarations, the components of a record
    type include any components declared by discriminant specifications of the
    record type declaration. The identifiers of all components of a record type must
    be distinct. The use of a name that denotes a record component other than a
    discriminant is not allowed within the record type definition that declares the
    component.

4   A component declaration with several identifiers is equivalent to a sequence
    of single component declarations, as explained in section 3.2. Each single
    component declaration declares a record component whose subtype is specified
    by the component subtype definition.

5   If a component declaration includes the assignment compound delimiter
    followed by an expression, the expression is the default expression of the
    record component; the default expression must be of the type of the component.
    Default expressions are not allowed for components that are of a limited type.

6   If a record type does not have a discriminant part, the same components are
    present in all values of the type. If the component list of a record type is
    defined by the reserved word **null** and there is no discriminant part, then the
    record type has no components and all records of the type are *null* records.

7   The elaboration of a record type definition creates a record type; it consists of
    the elaboration of any corresponding (single) component declarations, in the
    order in which they appear, including any component declaration in a variant
    part. The elaboration of a component declaration consists of the elaboration of
    the component subtype definition.

8   For the elaboration of a component subtype definition, if the constraint does
    not depend on a discriminant (see 3.7.1), then the subtype indication is
    elaborated. If, on the other hand, the constraint depends on a discriminant,
    then the elaboration consists of the evaluation of any included expression that
    is not a discriminant. [33]

---

[33] See also Appendix G, AI-00358.

9 **Examples of record type declarations:**

```
type DATE is
   record
      DAY   : INTEGER range 1 .. 31;
      MONTH : MONTH_NAME;
      YEAR  : INTEGER range 0 .. 4000;
   end record;

type COMPLEX is
   record
      RE : REAL := 0.0;
      IM : REAL := 0.0;
   end record;
```

10 **Examples of record variables:**

```
TOMORROW, YESTERDAY : DATE;
A, B, C : COMPLEX;

- both components of A, B, and C
-- are implicitly initialized to zero
```

Notes:

11 The default expression of a record component is implicitly evaluated by the elaboration of the declaration of a record object, in the absence of an explicit initialization (see 3.2.1). If a component declaration has several identifiers, the expression is evaluated once for each such component of the object (since the declaration is equivalent to a sequence of single component declarations).

12 Unlike the components of an array, the components of a record need not be of the same type.

13 **References:** assignment compound delimiter 2.2, component 3.3, composite value 3.3, constraint 3.3, declaration 3.1, depend on a discriminant 3.7.1, discriminant 3.3, discriminant part 3.7 3.7.1, elaboration 3.9, expression 4.4, identifier 2.3, identifier list 3.2, limited type 7.4.4, name 4.1, object 3.2, subtype 3.3, type 3.3, type mark 3.3.2, variant part 3.7.3

3.7.1 Discriminants

1 A discriminant part specifies the discriminants of a type. A discriminant of a record is a component of the record. The type of a discriminant must be discrete.

```
2    discriminant_part ::=
         (discriminant_specification {; discriminant_specification})

     discriminant_specification ::=
         identifier_list : type_mark [:= expression]
```

3   A discriminant part is only allowed in the type declaration for a record type, in a private type declaration or an incomplete type declaration (the corresponding full declaration must then declare a record type), and in the generic parameter declaration for a formal private type.

4   A discriminant specification with several identifiers is equivalent to a sequence of single discriminant specifications, as explained in section 3.2. Each single discriminant specification declares a discriminant. If a discriminant specification includes the assignment compound delimiter followed by an expression, the expression is the default expression of the discriminant; the default expression must be of the type of the discriminant. Default expressions must be provided either for all or for none of the discriminants of a discriminant part.

5   The use of the name of a discriminant is not allowed in default expressions of a discriminant part if the specification of the discriminant is itself given in the discriminant part.

6   Within a record type definition the only allowed uses of the name of a discriminant of the record type are: in the default expressions for record components; in a variant part as the discriminant name; and in a component subtype definition, either as a bound in an index constraint, or to specify a discriminant value in a discriminant constraint. A discriminant name used in these component subtype definitions must appear by itself, not as part of a larger expression. Such component subtype definitions and such constraints are said to *depend on a discriminant.*

7   A component is said to *depend on a discriminant* if it is a record component declared in a variant part, or a record component whose component subtype definition depends on a discriminant, or finally, one of the subcomponents of a component that itself depends on a discriminant.

8   Each record value includes a value for each discriminant specified for the record type; it also includes a value for each record component that does not depend on a discriminant. The values of the discriminants determine which other component values are in the record value.

9   Direct assignment to a discriminant of an object is not allowed; furthermore a discriminant is not allowed as an actual parameter of mode **in out** or **out**, or as a generic actual parameter of mode **in out.** The only allowed way to change the value of a discriminant of a variable is to assign a (complete) value to the variable itself. Similarly, an assignment to the variable itself is the only allowed way to change the constraint of one of its components, if the component subtype definition depends on a discriminant of the variable.

10 The elaboration of a discriminant part has no other effect.

₁₁ **Examples:**

```
type BUFFER(SIZE : BUFFER_SIZE := 100) is       -- see 3.5.4
   record
      POS   : BUFFER_SIZE := 0;
      VALUE : STRING(1 .. SIZE);
   end record;

type SQUARE(SIDE : INTEGER) is
   record
      MAT : MATRIX(1 .. SIDE, 1 .. SIDE);       -- see 3.6
   end record;

type DOUBLE_SQUARE(NUMBER : INTEGER) is
   record
      LEFT  : SQUARE(NUMBER);
      RIGHT : SQUARE(NUMBER);
   end record;

type ITEM(NUMBER : POSITIVE) is
   record
      CONTENT : INTEGER;
      --  no component depends on the discriminant
   end record;
```

₁₂ **References:** assignment 5.2, assignment compound delimiter 2.2, bound of a range 3.5, component 3.3, component declaration 3.7, component of a record 3.7, declaration 3.1, discrete type 3.5, discriminant 3.3, discriminant constraint 3.7.2, elaboration 3.9, expression 4.4, generic formal type 12.1, generic parameter declaration 12.1, identifier 2.3, identifier list 3.2, incomplete type declaration 3.8.1, index constraint 3.6.1, name 4.1, object 3.2, private type 7.4, private type declaration 7.4, record type 3.7, scope 8.2, simple name 4.1, subcomponent 3.3, subtype indication 3.3.2, type declaration 3.3.1, type mark 3.3.2, variant part 3.7.3

3.7.2 Discriminant Constraints

₁ A discriminant constraint is only allowed in a subtype indication, after a type mark. This type mark must denote either a type with discriminants, or an access type whose designated type is a type with discriminants. A discriminant constraint specifies the values of these discriminants.

₂
```
discriminant_constraint ::=
   (discriminant_association {, discriminant_association})

discriminant_association ::=
   [discriminant_simple_name
      {| discriminant_simple_name} =>] expression
```

₃ Each discriminant association associates an expression with one or more discriminants. A discriminant association is said to be *named* if the discriminants are specified explicitly by their names; it is otherwise said to be *positional.* For a positional association, the (single) discriminant is implicitly specified by position, in textual order. Named associations can be

given in any order, but if both positional and named associations are used in the same discriminant constraint, then positional associations must occur first, at their normal position. Hence once a named association is used, the rest of the discriminant constraint must use only named associations.

4   For a named discriminant association, the discriminant names must denote discriminants of the type for which the discriminant constraint is given. A discriminant association with more than one discriminant name is only allowed if the named discriminants are all of the same type. Furthermore, for each discriminant association (whether named or positional), the expression and the associated discriminants must have the same type. A discriminant constraint must provide exactly one value for each discriminant of the type.

5   A discriminant constraint is compatible with the type denoted by a type mark, if and only if each discriminant value belongs to the subtype of the corresponding discriminant. In addition, for each subcomponent whose component subtype specification depends on a discriminant, the discriminant value is substituted for the discriminant in this component subtype specification and the compatibility of the resulting subtype indication is checked. [34]

6   A composite value satisfies a discriminant constraint if and only if each discriminant of the composite value has the value imposed by the discriminant constraint.

7   The initial values of the discriminants of an object of a type with discriminants are determined as follows:

8   • For a variable declared by an object declaration, the subtype indication of the corresponding object declaration must impose a discriminant constraint unless default expressions exist for the discriminants; the discriminant values are defined either by the constraint or, in its absence, by the default expressions. The same requirement exists for the subtype indication of a component declaration, if the type of the record component has discriminants; and for the component subtype indication of an array type, if the type of the array components is a type with discriminants. [35]

9   • For a constant declared by an object declaration, the values of the discriminants are those of the initial value if the subtype of the constant is unconstrained; they are otherwise defined by this subtype (in the latter case, an exception is raised if the initial value does not

---

[34] See also Appendix G, AI-00007, AI-00319, and AI-00358.
[35] See also Appendix G, AI-00014.

belong to this subtype). The same rule applies to a generic parameter of mode **in.**

10 • For an object designated by an access value, the discriminant values must be defined by the allocator that creates the object. (The allocated object is constrained with the corresponding discriminant values.)

11 • For a formal parameter of a subprogram or entry, the discriminants of the formal parameter are initialized with those of the corresponding actual parameter. (The formal parameter is constrained if the corresponding actual parameter is constrained, and in any case if the mode is **in** or if the subtype of the formal parameter is constrained.)

12 • For a renaming declaration and for a generic formal parameter of mode **in out**, the discriminants are those of the renamed object or of the corresponding generic actual parameter.

13 For the elaboration of a discriminant constraint, the expressions given in the discriminant associations are evaluated in some order that is not defined by the language; the expression of a named association is evaluated once for each named discriminant.

14 **Examples using types declared in the previous section:**

```
LARGE   : BUFFER(200);   --  constrained, always 200 characters
                         --  (explicit discriminant value)

MESSAGE : BUFFER;        --  unconstrained, initially 100
                         --  characters (default discriminant
                         --  value)

BASIS   : SQUARE(5);     --  constrained, always 5 by 5

ILLEGAL : SQUARE;        --  illegal, a SQUARE must be
                         --  constrained
```

Note:

15 The above rules and the rules defining the elaboration of an object declaration (see 3.2) ensure that discriminants always have a value. In particular, if a discriminant constraint is imposed on an object declaration, each discriminant is initialized with the value specified by the constraint. Similarly, if the subtype of a component has a discriminant constraint, the discriminants of the component are correspondingly initialized.

16 **References:** access type 3.8, access type definition 3.8, access value 3.8, actual parameter 6.4.1, allocator 4.8, array type definition 3.6, bound of a range 3.5, compatible 3.3.2, component 3.3, component declaration 3.7, component subtype indication 3.7, composite value 3.3, constant 3.2.1, constrained subtype 3.3, constraint 3.3, declaration 3.1, default expression for a discriminant 3.7, depend on a discriminant 3.7.1, designate 3.8, designated type 3.8, discriminant 3.3, elaboration 3.9, entry 9.5,

evaluation 4.5, expression 4.4, formal parameter 6.1, generic actual parameter 12.3, generic formal parameter 12.1 12.3, mode in 6.1, mode in out 6.1, name 4.1, object 3.2, object declaration 3.2.1, renaming declaration 8.5, reserved word 2.9, satisfy 3.3, simple name 4.1, subcomponent 3.3, subprogram 6, subtype 3.3, subtype indication 3.3.2, type 3.3, type mark 3.3.2, variable 3.2.1

3.7.3 Variant Parts

1   A record type with a variant part specifies alternative lists of components. Each variant defines the components for the corresponding value or values of the discriminant.

```
2       variant_part ::=
            case discriminant_simple_name is
                variant
                {variant}
            end case;

        variant ::=
            when choice {| choice} =>
                component_list

        choice ::= simple_expression
            | discrete_range | others | component_simple_name
```

3   Each variant starts with a list of choices which must be of the same type as the discriminant of the variant part. The type of the discriminant of a variant part must not be a generic formal type. If the subtype of the discriminant is static, then each value of this subtype must be represented once and only once in the set of choices of the variant part, and no other value is allowed. Otherwise, each value of the (base) type of the discriminant must be represented once and only once in the set of choices.

4   The simple expressions and discrete ranges given as choices in a variant part must be static. A choice defined by a discrete range stands for all values in the corresponding range (none if a null range). The choice **others** is only allowed for the last variant and as its only choice; it stands for all values (possibly none) not given in the choices of previous variants. A component simple name is not allowed as a choice of a variant (although it is part of the syntax of choice).

5   A record value contains the values of the components of a given variant if and only if the discriminant value is equal to one of the values specified by the choices of the variant. This rule applies in turn to any further variant that is, itself, included in the component list of the given variant. If the component list of a variant is specified by **null**, the variant has no components.

6   **Example of record type with a variant part:**

```
type DEVICE is (PRINTER, DISK, DRUM);
type STATE  is (OPEN, CLOSED);

type PERIPHERAL(UNIT : DEVICE := DISK) is
   record
      STATUS : STATE;
      case UNIT is
         when PRINTER =>
            LINE_COUNT : INTEGER range 1 .. PAGE_SIZE;
         when others =>
            CYLINDER   : CYLINDER_INDEX;
            TRACK      : TRACK_NUMBER;
      end case;
   end record;
```

7   **Examples of record subtypes:**

```
subtype DRUM_UNIT is PERIPHERAL(DRUM);
subtype DISK_UNIT is PERIPHERAL(DISK);
```

8   **Examples of constrained record variables:**

```
WRITER  : PERIPHERAL(UNIT => PRINTER);
ARCHIVE : DISK_UNIT;
```

Note:

9   Choices with discrete values are also used in case statements and in array aggregates. Choices with component simple names are used in record aggregates.

10 **References:** array aggregate 4.3.2, base type 3.3, component 3.3, component list 3.7, discrete range 3.6, discriminant 3.3, generic formal type 12.1.2, null range 3.5, record aggregate 4.3.1, range 3.5, record type 3.7, simple expression 4.4, simple name 4.1, static discrete range 4.9, static expression 4.9, static subtype 4.9, subtype 3.3

3.7.4 Operations of Record Types

1   The basic operations of a record type include the operations involved in assignment and aggregates (unless the type is limited), membership tests, selection of record components, qualification, and type conversion (for derived types).

2   For any object A of a type with discriminants, the basic operations also include the following attribute:

3   A′CONSTRAINED   Yields the value TRUE if a discriminant constraint applies to the object A, or if the object is a constant (including a formal parameter or generic formal parameter of mode **in**); yields the value FALSE otherwise. If A is a generic formal parameter of mode **in out**, or if A is a formal parameter of mode

> **in out** or **out** and the type mark given in the
> corresponding parameter specification denotes an
> unconstrained type with discriminants, then the
> value of this attribute is obtained from that of the
> corresponding actual parameter. The value of this
> attribute is of the predefined type BOOLEAN.

4 In addition, the attributes T'BASE and T'SIZE are defined for a record type
   or subtype T (see 3.3.3); the attributes A'SIZE and A'ADDRESS are defined
   for a record object A (see 13.7.2).

5 Besides the basic operations, the operations of a record type include the
   predefined comparison for equality and inequality, unless the type is limited.

   Note:

6 A record type is limited if the type of any of its components is limited (see
   7.4.4).

7 **References:** actual parameter 6.4.1, aggregate 4.3, assignment 5.2, attribute 4.1.4,
   basic operation 3.3.3, boolean type 3.5.3, constant 3.2.1, conversion 4.6, derived
   type 3.4, discriminant 3.3, discriminant constraint 3.7.2, formal parameter 6.1,
   generic actual parameter 12.3, generic formal parameter 12.1 12.3, limited type 7.4.4,
   membership test 4.5 4.5.2, mode 6.1, object 3.2.1, operation 3.3, predefined operator
   4.5, predefined type C, qualified expression 4.7, record type 3.7, relational operator 4.5
   4.5.2, selected component 4.1.3, subcomponent 3.3, subtype 3.3, type 3.3

3.8 Access Types

1 An object declared by an object declaration is created by the elaboration of
   the object declaration and is denoted by a simple name or by some other form
   of name. In contrast, there are objects that are created by the evaluation of
   allocators (see 4.8) and that have no simple name. Access to such an object is
   achieved by an *access value* returned by an allocator; the access value is said
   to *designate* the object.

2   access_type_definition ::= **access** subtype_indication

3 For each access type, there is a literal **null** which has a null access value
   designating no object at all. The null value of an access type is the default
   initial value of the type. Other values of an access type are obtained by
   evaluation of a special operation of the type, called an allocator. Each such
   access value designates an object of the subtype defined by the subtype
   indication of the access type definition; this subtype is called the *designated
   subtype*; the base type of this subtype is called the *designated type.* The
   objects designated by the values of an access type form a *collection* implicitly
   associated with the type.

4 The elaboration of an access type definition consists of the elaboration of the subtype indication and creates an access type.

5 If an access object is constant, the contained access value cannot be changed and always designates the same object. On the other hand, the value of the designated object need not remain the same (assignment to the designated object is allowed unless the designated type is limited).

6 The only forms of constraint that are allowed after the name of an access type in a subtype indication are index constraints and discriminant constraints. (See sections 3.6.1 and 3.7.2 for the rules applicable to these subtype indications.) An access value *belongs* to a corresponding subtype of an access type either if the access value is the null value or if the value of the designated object satisfies the constraint. [36]

7 **Examples:**

```
type FRAME is access MATRIX;          -- see 3.6
type BUFFER_NAME is access BUFFER;  -- see 3.7.1
```

Notes:

8 An access value delivered by an allocator can be assigned to several access objects. Hence it is possible for an object created by an allocator to be designated by more than one variable or constant of the access type. An access value can only designate an object created by an allocator; in particular, it cannot designate an object declared by an object declaration.

9 If the type of the objects designated by the access values is an array type or a type with discriminants, these objects are constrained with either the array bounds or the discriminant values supplied implicitly or explicitly for the corresponding allocators (see 4.8).

10 Access values are called *pointers* or *references* in some other languages.

11 **References:** allocator 4.8, array type 3.6, assignment 5.2, belong to a subtype 3.3, constant 3.2.1, constraint 3.3, discriminant constraint 3.7.2, elaboration 3.9, index constraint 3.6.1, index specification 3.6, limited type 7.4.4, literal 4.2, name 4.1, object 3.2.1, object declaration 3.2.1, reserved word 2.9, satisfy 3.3, simple name 4.1, subcomponent 3.3, subtype 3.3, subtype indication 3.3.2, type 3.3, variable 3.2.1

---

[36] See also Appendix G, AI-00324.

3.8.1 Incomplete Type Declarations

1   There are no particular limitations on the designated type of an access type. In particular, the type of a component of the designated type can be another access type, or even the same access type. This permits mutually dependent and recursive access types. Their declarations require a prior incomplete (or private) type declaration for one or more types.

2
```
incomplete_type_declaration ::=
    type identifier [discriminant_part];
```

3   For each incomplete type declaration, there must be a corresponding declaration of a type with the same identifier. The corresponding declaration must be either a full type declaration or the declaration of a task type. In the rest of this section, explanations are given in terms of full type declarations; the same rules apply also to declarations of task types. If the incomplete type declaration occurs immediately within either a declarative part or the visible part of a package specification, then the full type declaration must occur later and immediately within this declarative part or visible part. If the incomplete type declaration occurs immediately within the private part of a package, then the full type declaration must occur later and immediately within either the private part itself, or the declarative part of the corresponding package body.

4   A discriminant part must be given in the full type declaration if and only if one is given in the incomplete type declaration; if discriminant parts are given, then they must conform (see 6.3.1 for the conformance rules). Prior to the end of the full type declaration, the only allowed use of a name that denotes a type declared by an incomplete type declaration is as the type mark in the subtype indication of an access type definition; the only form of constraint allowed in this subtype indication is a discriminant constraint. [37]

5   The elaboration of an incomplete type declaration creates a type. If the incomplete type declaration has a discriminant part, this elaboration includes that of the discriminant part: in such a case, the discriminant part of the full type declaration is not elaborated.

---

[37] See also Appendix G, AI-00007, AI-00231, and AI-00319.

6 **Example of a recursive type:**

```
type CELL;  --  incomplete type declaration
type LINK is access CELL;

type CELL is
   record
      VALUE : INTEGER;
      SUCC  : LINK;
      PRED  : LINK;
   end record;

HEAD : LINK := new CELL'(0, null, null);
NEXT : LINK := HEAD.SUCC;
```

7 **Examples of mutually dependent access types:**

```
type PERSON(SEX : GENDER);  --  incomplete type declaration
type CAR;                   --  incomplete type declaration

type PERSON_NAME is access PERSON;
type CAR_NAME    is access CAR;

type CAR is
   record
      NUMBER : INTEGER;
      OWNER  : PERSON_NAME;
   end record;

type PERSON(SEX : GENDER) is
   record
      NAME    : STRING(1 .. 20);
      BIRTH   : DATE;
      AGE     : INTEGER range 0 .. 130;
      VEHICLE : CAR_NAME;
      case SEX is
         when M => WIFE    : PERSON_NAME(SEX => F);
         when F => HUSBAND : PERSON_NAME(SEX => M);
      end case;
   end record;

MY_CAR, YOUR_CAR, NEXT_CAR : CAR_NAME;  -- implicitly
                                        -- initialized
                                        -- with null value
```

8 **References:** access type 3.8, access type definition 3.8, component 3.3, conform 6.3.1, constraint 3.3, declaration 3.1, declarative item 3.9, designate 3.8, discriminant constraint 3.7.2, discriminant part 3.7.1, elaboration 3.9, identifier 2.3, name 4.1, subtype indication 3.3.2, type 3.3, type mark 3.3.2

3.8.2 Operations of Access Types

1   The basic operations of an access type include the operations involved in assignment, allocators for the access type, membership tests, qualification, explicit conversion, and the literal **null.** If the designated type is a type with discriminants, the basic operations include the selection of the corresponding discriminants; if the designated type is a record type, they include the selection of the corresponding components; if the designated type is an array type, they include the formation of indexed components and slices; if the designated type is a task type, they include selection of entries and entry families. Furthermore, the basic operations include the formation of a selected component with the reserved word **all** (see 4.1.3).

2   If the designated type is an array type, the basic operations include the attributes that have the attribute designators FIRST, LAST, RANGE, and LENGTH (likewise, the attribute designators of the N-th dimension). The prefix of each of these attributes must be a value of the access type. These attributes yield the corresponding characteristics of the designated object (see 3.6.2).

3   If the designated type is a task type, the basic operations include the attributes that have the attribute designators TERMINATED and CALLABLE (see 9.9). The prefix of each of these attributes must be a value of the access type. These attributes yield the corresponding characteristics of the designated task objects.

4   In addition, the attribute T$'$BASE (see 3.3.3) and the representation attributes T$'$SIZE and T$'$STORAGE_SIZE (see 13.7.2) are defined for an access type or subtype T; the attributes A$'$SIZE and A$'$ADDRESS are defined for an access object A (see 13.7.2).

5   Besides the basic operations, the operations of an access type include the predefined comparison for equality and inequality.

6   **References:** access type 3.8, allocator 4.8, array type 3.6, assignment 5.2, attribute 4.1.4, attribute designator 4.1.4, base type 3.3, basic operation 3.3.3, collection 3.8, constrained array subtype 3.6, conversion 4.6, designate 3.8, designated subtype 3.8, designated type 3.8, discriminant 3.3, indexed component 4.1.1, literal 4.2, membership test 4.5 4.5.2, object 3.2.1, operation 3.3, private type 7.4, qualified expression 4.7, record type 3.7, selected component 4.1.3, slice 4.1.2, subtype 3.3, task type 9.1, type 3.3

3.9 Declarative Parts

1 A declarative part contains declarative items (possibly none).

2
```
declarative_part ::=
    {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::= basic_declaration
    | representation_clause | use_clause

later_declarative_item ::= body
    | subprogram_declaration | package_declaration
    | task_declaration        | generic_declaration
    | use_clause              | generic_instantiation

body ::= proper_body | body_stub

proper_body ::= subprogram_body | package_body | task_body
```

3 The elaboration of a declarative part consists of the elaboration of the declarative items, if any, in the order in which they are given in the declarative part. After its elaboration, a declarative item is said to be *elaborated.* Prior to the completion of its elaboration (including before the elaboration), the declarative item is not yet elaborated.

4 For several forms of declarative item, the language rules (in particular scope and visibility rules) are such that it is either impossible or illegal to use an entity before the elaboration of the declarative item that declares this entity. For example, it is not possible to use the name of a type for an object declaration if the corresponding type declaration is not yet elaborated. In the case of bodies, the following checks are performed:

5 • For a subprogram call, a check is made that the body of the subprogram is already elaborated. [38]

6 • For the activation of a task, a check is made that the body of the corresponding task unit is already elaborated. [39]

7 • For the instantiation of a generic unit that has a body, a check is made that this body is already elaborated.

8 The exception PROGRAM_ERROR is raised if any of these checks fails. [40]

---

[38] See also Appendix G, AI-00180 and AI-00406.
[39] See also Appendix G, AI-00149.
[40] See also Appendix G, AI-00430.

9   If a subprogram declaration, a package declaration, a task declaration, or a generic declaration is a declarative item of a given declarative part, then the body (if there is one) of the program unit declared by the declarative item must itself be a declarative item of this declarative part (and must appear later). If the body is a body stub, then a separately compiled subunit containing the corresponding proper body is required for the program unit (see 10.2).

10   **References:** activation 9.3, instantiation 12.3, program_error exception 11.1, scope 8.2, subprogram call 6.4, type 3.3, visibility 8.3

11   **Elaboration of declarations:** 3.1, component declaration 3.7, deferred constant declaration 7.4.3, discriminant specification 3.7.1, entry declaration 9.5, enumeration literal specification 3.5.1, generic declaration 12.1, generic instantiation 12.3, incomplete type declaration 3.8.1, loop parameter specification 5.5, number declaration 3.2.2, object declaration 3.2.1, package declaration 7.2, parameter specification 6.1, private type declaration 7.4.1, renaming declaration 8.5, subprogram declaration 6.1, subtype declaration 3.3.2, task declaration 9.1, type declaration 3.3.1

12   **Elaboration of type definitions:** 3.3.1, access type definition 3.8, array type definition 3.6, derived type definition 3.4, enumeration type definition 3.5.1, integer type definition 3.5.4, real type definition 3.5.6, record type definition 3.7

13   **Elaboration of other constructs:** context clause 10.1, body stub 10.2, compilation unit 10.1, discriminant part 3.7.1, generic body 12.2, generic formal parameter 12.1 12.3, library unit 10.5, package body 7.1, representation clause 13.1, subprogram body 6.3, subunit 10.2, task body 9.1, task object 9.2, task specification 9.1, use clause 8.4, with clause 10.1.1

4

Names and Expressions

1　The rules applicable to the different forms of name and expression, and to their evaluation, are given in this chapter.

4.1 Names

1　Names can denote declared entities, whether declared explicitly or implicitly (see 3.1). Names can also denote objects designated by access values; subcomponents and slices of objects and values; single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing.

2
```
name ::= simple_name
    | character_literal  | operator_symbol
    | indexed_component  | slice
    | selected_component | attribute

simple_name ::= identifier

prefix ::= name | function_call
```

3　A simple name for an entity is either the identifier associated with the entity by its declaration, or another identifier associated with the entity by a renaming declaration.

4　Certain forms of name (indexed and selected components, slices, and attributes) include a *prefix* that is either a name or a function call. If the type of a prefix is an access type, then the prefix must not be a name that denotes a formal parameter of mode **out** or a subcomponent thereof.

5　If the prefix of a name is a function call, then the name denotes a component, a slice, an attribute, an entry, or an entry family, either of the result of the function call, or (if the result is an access value) of the object designated by the result.

6　A prefix is said to be *appropriate for a type* in either of the following cases:

7　　• The type of the prefix is the type considered.

8   •   The type of the prefix is an access type whose designated type is the type considered.

9   The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a simple name, a character literal, or an operator symbol.

10   The evaluation of a name that has a prefix includes the evaluation of the prefix, that is, of the corresponding name or function call. If the type of the prefix is an access type, the evaluation of the prefix includes the determination of the object designated by the corresponding access value; the exception CONSTRAINT_ERROR is raised if the value of the prefix is a null access value, except in the case of the prefix of a representation attribute (see 13.7.2).

11   **Examples of simple names:**
```
PI      --  the simple name of a number          (see 3.2.2)
LIMIT   --  the simple name of a constant         (see 3.2.1)
COUNT   --  the simple name of a scalar variable  (see 3.2.1)
BOARD   --  the simple name of an array variable  (see 3.6.1)
MATRIX  --  the simple name of a type             (see 3.6)
RANDOM  --  the simple name of a function         (see 6.1)
ERROR   --  the simple name of an exception       (see 11.1)
```

12   **References:** access type 3.8, access value 3.8, attribute 4.1.4, belong to a type 3.3, character literal 2.5, component 3.3, constraint_error exception 11.1, declaration 3.1, designate 3.8, designated type 3.8, entity 3.1, entry 9.5, entry family 9.5, evaluation 4.5, formal parameter 6.1, function call 6.4, identifier 2.3, indexed component 4.1.1, mode 6.1, null access value 3.8, object 3.2.1, operator symbol 6.1, raising of exceptions 11, renaming declarations 8.5, selected component 4.1.3, slice 4.1.2, subcomponent 3.3, type 3.3

4.1.1 Indexed Components

1   An indexed component denotes either a component of an array or an entry in a family of entries.

2   `indexed_component ::= prefix(expression {, expression})`

3   In the case of a component of an array, the prefix must be appropriate for an array type. The expressions specify the index values for the component; there must be one such expression for each index position of the array type. In the case of an entry in a family of entries, the prefix must be a name that denotes an entry family of a task object, and the expression (there must be exactly one) specifies the index value for the individual entry.

4   Each expression must be of the type of the corresponding index. For the
     evaluation of an indexed component, the prefix and the expressions are
     evaluated in some order that is not defined by the language. The exception
     CONSTRAINT_ERROR is raised if an index value does not belong to the range
     of the corresponding index of the prefixing array or entry family.

5   **Examples of indexed components:**

```
MY_SCHEDULE(SAT)    -- a component of a              (see 3.6.1)
                    -- one-dimensional array

PAGE(10)            -- a component of a              (see 3.6)
                    -- one-dimensional array

BOARD(M, J + 1)     -- a component of a              (see 3.6.1)
                    -- two-dimensional array

PAGE(10)(20)        -- a component of a component    (see 3.6)
REQUEST(MEDIUM)     -- an entry in a family of entries (see 9.5)
NEXT_FRAME(L)(M, N) -- a component of a function call  (see 6.1)
```

    Notes on the examples:

6   Distinct notations are used for components of multidimensional arrays (such
     as BOARD) and arrays of arrays (such as PAGE). The components of an array
     of arrays are arrays and can therefore be indexed. Thus PAGE(10)(20) denotes
     the 20th component of PAGE(10). In the last example NEXT_FRAME(L) is a
     function call returning an access value which designates a two-dimensional
     array.

7   **References:** appropriate for a type 4.1, array type 3.6, component 3.3, component of
     an array 3.6, constraint_error exception 11.1, dimension 3.6, entry 9.5, entry family
     9.5, evaluation 4.5, expression 4.4, function call 6.4, in some order 1.6, index 3.6, name
     4.1, prefix 4.1, raising of exceptions 11, returned value 5.8 6.5, task object 9.2

4.1.2 Slices

1   A slice denotes a one-dimensional array formed by a sequence of consecutive
     components of a one-dimensional array. A slice of a variable is a variable; a
     slice of a constant is a constant; a slice of a value is a value.

2   slice ::= prefix(discrete_range)

3   The prefix of a slice must be appropriate for a one-dimensional array type. The
     type of the slice is the base type of this array type. The bounds of the discrete
     range define those of the slice and must be of the type of the index; the slice is
     a *null slice* denoting a null array if the discrete range is a null range.

4 For the evaluation of a name that is a slice, the prefix and the discrete range are evaluated in some order that is not defined by the language. The exception CONSTRAINT_ERROR is raised by the evaluation of a slice, other than a null slice, if any of the bounds of the discrete range does not belong to the index range of the prefixing array. (The bounds of a null slice need not belong to the subtype of the index.)

5 **Examples of slices:**

```
STARS(1 .. 15)        -- a slice of        (see 3.6.3)
                      -- 15 characters
PAGE(10 .. 10 + SIZE) -- a slice of        (see 3.6 and 3.2.1)
                      -- 1 + SIZE
                      -- components
PAGE(L)(A .. B)       -- a slice of        (see 3.6)
                      -- the array PAGE(L)
STARS(1 .. 0)         -- a null slice      (see 3.6.3)
MY_SCHEDULE(WEEKDAY)  -- bounds given      (see 3.6 and 3.5.1)
                      -- by subtype
STARS(5 .. 15)(K)     -- same as STARS(K)  (see 3.6.3)
                      -- provided that K
                      -- is in 5 .. 15
```

Notes:

6 For a one-dimensional array A, the name A(N .. N) is a slice of one component; its type is the base type of A. On the other hand, A(N) is a component of the array A and has the corresponding component type.

7 **References:** appropriate for a type 4.1, array 3.6, array type 3.6, array value 3.8, base type 3.3, belong to a subtype 3.3, bound of a discrete range 3.6.1, component 3.3, component type 3.3, constant 3.2.1, constraint 3.3, constraint_error exception 11.1, dimension 3.6, discrete range 3.6, evaluation 4.5, index 3.6, index range 3.6, name 4.1, null array 3.6.1, null range 3.5, prefix 4.1, raising of exceptions 11, type 3.3, variable 3.2.1

4.1.3 Selected Components

1 Selected components are used to denote record components, entries, entry families, and objects designated by access values; they are also used as *expanded names* as described below.

2
```
selected_component ::= prefix.selector

selector ::= simple_name
   | character_literal | operator_symbol | all
```

3   The following four forms of selected components are used to denote a discriminant, a record component, an entry, or an object designated by an access value:

4   (a)   A discriminant:

5   The selector must be a simple name denoting a discriminant of an object or value. The prefix must be appropriate for the type of this object or value.

6   (b)   A component of a record:

7   The selector must be a simple name denoting a component of a record object or value. The prefix must be appropriate for the type of this object or value.

8   For a component of a variant, a check is made that the values of the discriminants are such that the record has this component. The exception CONSTRAINT_ERROR is raised if this check fails.

9   (c)   A single entry or an entry family of a task:

10   The selector must be a simple name denoting a single entry or an entry family of a task. The prefix must be appropriate for the type of this task.

11   (d)   An object designated by an access value:

12   The selector must be the reserved word **all.** The value of the prefix must belong to an access type.

13   A selected component of one of the remaining two forms is called an *expanded name.* In each case the selector must be either a simple name, a character literal, or an operator symbol. A function call is not allowed as the prefix of an expanded name. An expanded name can denote:

14   (e)   An entity declared in the visible part of a package:

15   The prefix must denote the package. The selector must be the simple name, character literal, or operator symbol of the entity.[1]

16   (f)   An entity whose declaration occurs immediately within a named construct:

---

[1]   See also Appendix G, AI-00016, AI-00187, AI-00412, and AI-00468.

17 The prefix must denote a construct that is either a program unit, a block statement, a loop statement, or an accept statement. In the case of an accept statement, the prefix must be either the simple name of the entry or entry family, or an expanded name ending with such a simple name (that is, no index is allowed). The selector must be the simple name, character literal, or operator symbol of an entity whose declaration occurs immediately within the construct.

18 This form of expanded name is only allowed within the construct itself (including the body and any subunits, in the case of a program unit). A name declared by a renaming declaration is not allowed as the prefix. If the prefix is the name of a subprogram or accept statement and if there is more than one visible enclosing subprogram or accept statement of this name, the expanded name is ambiguous, independently of the selector.[2]

19 If, according to the visibility rules, there is at least one possible interpretation of the prefix of a selected component as the name of an enclosing subprogram or accept statement, then the only interpretations considered are those of rule (f), as expanded names (no interpretations of the prefix as a function call are then considered).

20 The evaluation of a name that is a selected component includes the evaluation of the prefix.

21 **Examples of selected components:**

```
TOMORROW.MONTH      -- a record component    (see 3.7)
NEXT_CAR.OWNER      -- a record component    (see 3.8.1)
NEXT_CAR.OWNER.AGE --  a record component    (see 3.8.1)

WRITER.UNIT         -- a record component
                    -- (a discriminant)      (see 3.7.3)

MIN_CELL(H).VALUE   -- a record component
                    -- of the result of
                    -- the function call
                    -- MIN_CELL(H)            (see 6.1 and 3.8.1)

CONTROL.SEIZE       -- an entry of the
                    -- task CONTROL           (see 9.1 and 9.2)

POOL(K).WRITE       -- an entry of the
                    -- task POOL(K)           (see 9.1 and 9.2)

NEXT_CAR.ALL        -- the object designated
                    -- by the access variable
                    -- NEXT_CAR               (see 3.8.1)
```

---

[2] See also Appendix G, AI-00016 and AI-00412.

²² **Examples of expanded names:**

```
TABLE_MANAGER.INSERT --  a procedure of the
                     --  visible part of
                     --  a package             (see 7.5)

KEY_MANAGER."<"      --  an operator of the
                     --  visible part of       (see 7.4.2)
                     --  a package

DOT_PRODUCT.SUM      --  a variable declared
                     --  in a procedure body   (see 6.5)

BUFFER.POOL          --  a variable declared
                     --  in a task unit        (see 9.12)

BUFFER.READ          --  an entry of a task unit (see 9.12)

SWAP.TEMP            --  a variable declared in
                     --  a block statement     (see 5.6)

STANDARD.BOOLEAN     --  the name of a
                     --  predefined type       (see 8.6 and C)
```

Note:

²³ For a record with components that are other records, the above rules imply that the simple name must be given at each level for the name of a subcomponent. For example, the name NEXT_CAR.OWNER.BIRTH.MONTH cannot be shortened (NEXT_CAR.OWNER.MONTH is not allowed).

²⁴ **References:** accept statement 9.5, access type 3.8, access value 3.8, appropriate for a type 4.1, block statement 5.6, body of a program unit 3.9, character literal 2.5, component of a record 3.7, constraint_error exception 11.1, declaration 3.1, designate 3.8, discriminant 3.3, entity 3.1, entry 9.5, entry family 9.5, function call 6.4, index 3.6, loop statement 5.5, object 3.2.1, occur immediately within 8.1, operator 4.5, operator symbol 6.1, overloading 8.3, package 7, predefined type C, prefix 4.1, procedure body 6.3, program unit 6, raising of exceptions 11, record 3.7, record component 3.7, renaming declaration 8.5, reserved word 2.9, simple name 4.1, subprogram 6, subunit 10.2, task 9, task object 9.2, task unit 9, variable 3.7.3, variant 3.7.3, visibility 8.3, visible part 3.7.3

4.1.4 Attributes

¹ An attribute denotes a basic operation of an entity given by a prefix.

²
```
attribute ::= prefix'attribute_designator

attribute_designator ::=
   simple_name [(universal_static_expression)]
```

3   The applicable attribute designators depend on the prefix. An attribute can be a basic operation delivering a value; alternatively it can be a function, a type, or a range. The meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute.[3]

4   The attributes defined by the language are summarized in Annex A. In addition, an implementation may provide implementation-defined attributes; their description must be given in Appendix F. The attribute designator of any implementation-defined attribute must not be the same as that of any language-defined attribute.

5   The evaluation of a name that is an attribute consists of the evaluation of the prefix.

Notes:

6   The attribute designators DIGITS, DELTA, and RANGE have the same identifier as a reserved word. However, no confusion is possible since an attribute designator is always preceded by an apostrophe. The only predefined attribute designators that have a universal expression are those for certain operations of array types (see 3.6.2).

7   **Examples of attributes:**

```
COLOR'FIRST           -- minimum value of
                      -- the enumeration
                      -- type COLOR           (see 3.3.1 and 3.5)

RAINBOW'BASE'FIRST -- same as COLOR'FIRST   (see 3.3.2 and 3.3.3)

REAL'DIGITS           -- precision of the
                      -- type REAL            (see 3.5.7 and 3.5.8)

BOARD'LAST(2)         -- upper bound of the
                      -- second dimension of
                      -- BOARD                (see 3.6.1 and 3.6.2)

BOARD'RANGE(1)        -- index range of the
                      -- first dimension of
                      -- BOARD                (see 3.6.1 and 3.6.2)

POOL(K)'TERMINATED -- TRUE if task POOL(K)
                      -- is terminated        (see 9.2   and 9.9)

DATE'SIZE             -- number of bits for
                      -- records of type DATE (see 3.7   and 13.7.2)

MESSAGE'ADDRESS       -- address of the record
                      -- variable MESSAGE     (see 3.7.2 and 13.7.2)
```

---

[3] See also Appendix G, AI-00015.

8   **References:** appropriate for a type 4.1, basic operation 3.3.3, declared entity 3.1, name 4.1, prefix 4.1, reserved word 2.9, simple name 4.1, static expression 4.9, type 3.3, universal expression 4.10

4.2 Literals

1   A literal is either a numeric literal, an enumeration literal, the literal **null**, or a string literal. The evaluation of a literal yields the corresponding value.

2   Numeric literals are the literals of the types *universal_integer* and *universal_real.* Enumeration literals include character literals and yield values of the corresponding enumeration types. The literal **null** yields a null access value which designates no objects at all.

3   A string literal is a basic operation that combines a sequence of characters into a value of a one-dimensional array of a character type; the bounds of this array are determined according to the rules for positional array aggregates (see 4.3.2). For a null string literal, the upper bound is the predecessor, as given by the PRED attribute, of the lower bound. The evaluation of a null string literal raises the exception CONSTRAINT_ERROR if the lower bound does not have a predecessor (see 3.5.5).

4   The type of a string literal and likewise the type of the literal **null** must be determinable solely from the context in which this literal appears, excluding the literal itself, but using the fact that the literal **null** is a value of an access type, and similarly that a string literal is a value of a one-dimensional array type whose component type is a character type.

5   The character literals corresponding to the graphic characters contained within a string literal must be visible at the place of the string literal (although these characters themselves are not used to determine the type of the string literal).

6   **Examples:**
```
3.14159_26536  --  a real literal
1_345          --  an integer literal
CLUBS          --  an enumeration literal
'A'            --  a character literal
"SOME TEXT"    --  a string literal
```

7   **References:** access type 3.8, aggregate 4.3, array 3.6, array bound 3.6, array type 3.6, character literal 2.5, character type 3.5.2, component type 3.3, constraint_error exception 11.1, designate 3.8, dimension 3.6, enumeration literal 3.5.1, graphic character 2.1, integer literal 2.4, null access value 3.8, null literal 3.8, numeric literal 2.4, object 3.2.1, real literal 2.4, string literal 2.6, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6, visibility 8.3

4.3 Aggregates

1   An aggregate is a basic operation that combines component values into a
     composite value of a record or array type.

2   ```
     aggregate ::=
         (component_association {, component_association})

     component_association ::=
         [choice {| choice} =>] expression
     ```

3    Each component association associates an expression with components
     (possibly none). A component association is said to be *named* if the components
     are specified explicitly by choices; it is otherwise said to be *positional.* For
     a positional association, the (single) component is implicitly specified by
     position, in the order of the corresponding component declarations for record
     components, in index order for array components.

4    Named associations can be given in any order (except for the choice **others**),
     but if both positional and named associations are used in the same aggregate,
     then positional associations must occur first, at their normal position. Hence
     once a named association is used, the rest of the aggregate must use only
     named associations. Aggregates containing a single component association
     must always be given in named notation. Specific rules concerning component
     associations exist for record aggregates and array aggregates.

5    Choices in component associations have the same syntax as in variant parts
     (see 3.7.3). A choice that is a component simple name is only allowed in a
     record aggregate. For a component association, a choice that is a simple
     expression or a discrete range is only allowed in an array aggregate; a choice
     that is a simple expression specifies the component at the corresponding index
     value; similarly a discrete range specifies the components at the index values
     in the range. The choice **others** is only allowed in a component association
     if the association appears last and has this single choice; it specifies all
     remaining components, if any.

6    Each component of the value defined by an aggregate must be represented once
     and only once in the aggregate. Hence each aggregate must be complete and a
     given component is not allowed to be specified by more than one choice.[4]

7    The type of an aggregate must be determinable solely from the context in
     which the aggregate appears, excluding the aggregate itself, but using the fact
     that this type must be composite and not limited. The type of an aggregate in
     turn determines the required type for each of its components.

---

[4]   See also Appendix G, AI-00169 and AI-00293.

**Notes:**

8   The above rule implies that the determination of the type of an aggregate cannot use any information from within the aggregate. In particular, this determination cannot use the type of the expression of a component association, or the form or the type of a choice. An aggregate can always be distinguished from an expression enclosed by parentheses: this is a consequence of the fact that named notation is required for an aggregate with a single component.

9   **References:** array aggregate 4.3.2, array type 3.6, basic operation 3.3.3, choice 3.7.3, component 3.3, composite type 3.3, composite value 3.3, discrete range 3.6, expression 4.4, index 3.6, limited type 7.4.4, primary 4.4, record aggregate 4.3.1, record type 3.7, simple expression 4.4, simple name 4.1, type 3.3, variant part 3.7.3

## 4.3.1 Record Aggregates

1   If the type of an aggregate is a record type, the component names given as choices must denote components (including discriminants) of the record type. If the choice **others** is given as a choice of a record aggregate, it must represent at least one component. A component association with the choice **others** or with more than one choice is only allowed if the represented components are all of the same type. The expression of a component association must have the type of the associated record components.[5]

2   The value specified for a discriminant that governs a variant part must be given by a static expression (note that this value determines which dependent components must appear in the record value).

3   For the evaluation of a record aggregate, the expressions given in the component associations are evaluated in some order that is not defined by the language. The expression of a named association is evaluated once for each associated component. A check is made that the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent. The exception CONSTRAINT_ERROR is raised if this check fails.

4   **Example of a record aggregate with positional associations:**
```
(4, JULY, 1776)   -- see 3.7
```

5   **Examples of record aggregates with named associations:**
```
(DAY => 4, MONTH => JULY, YEAR => 1776)
(MONTH => JULY, DAY => 4, YEAR => 1776)

(DISK, CLOSED, TRACK => 5, CYLINDER => 12)   -- see 3.7.3
(UNIT => DISK, STATUS => CLOSED, CYLINDER => 9, TRACK => 1)
```

---

[5]  See also Appendix G, AI-00244.

6 **Example of component association with several choices:**

```
(VALUE => 0, SUCC|PRED => new CELL'(0, null, null)) -- see 3.8.1
-- The allocator is evaluated twice:
-- SUCC and PRED designate different cells
```

Note:

7 For an aggregate with positional associations, discriminant values appear first since the discriminant part is given first in the record type declaration; they must be in the same order as in the discriminant part.

8 **References:** aggregate 4.3, allocator 4.8, choice 3.7.3, component association 4.3, component name 3.7, constraint 3.3, constraint_error exception 11.1, depend on a discriminant 3.7.1, discriminant 3.3, discriminant part 3.7.1, evaluate 4.5, expression 4.4, in some order 1.6, program 10, raising of exceptions 11, record component 3.7, record type 3.7, satisfy 3.3, static expression 4.9, subcomponent 3.3, subtype 3.3.2, type 3.3, variant part 3.7.3

4.3.2 Array Aggregates

1 If the type of an aggregate is a one-dimensional array type, then each choice must specify values of the index type, and the expression of each component association must be of the component type.

2 If the type of an aggregate is a multidimensional array type, an n-dimensional aggregate is written as a one-dimensional aggregate, in which the expression specified for each component association is itself written as an $(n - 1)$-dimensional aggregate which is called a *subaggregate*; the index subtype of the one-dimensional aggregate is given by the first index position of the array type. The same rule is used to write a subaggregate if it is again multidimensional, using successive index positions. A string literal is allowed in a multidimensional aggregate at the place of a one-dimensional array of a character type. In what follows, the rules concerning array aggregates are formulated in terms of one-dimensional aggregates.

3 Apart from a final component association with the single choice **others**, the rest (if any) of the component associations of an array aggregate must be either all positional or all named. A named association of an array aggregate is only allowed to have a choice that is not static, or likewise a choice that is a null range, if the aggregate includes a single component association and this component association has a single choice. An **others** choice is static if the applicable index constraint is static.[6]

---

[6] See also Appendix G, AI-00190 and AI-00310.

4   The bounds of an array aggregate that has an **others** choice are determined by the applicable index constraint. An **others** choice is only allowed if the aggregate appears in one of the following contexts (which defines the applicable index constraint):

5   (a) The aggregate is an actual parameter, a generic actual parameter, the result expression of a function, or the expression that follows an assignment compound delimiter. Moreover, the subtype of the corresponding formal parameter, generic formal parameter, function result, or object is a constrained array subtype.

6   For an aggregate that appears in such a context and contains an association with an **others** choice, named associations are allowed for other associations only in the case of a (nongeneric) actual parameter or function result. If the aggregate is a multidimensional array, this restriction also applies to each of its subaggregates.

7   (b) The aggregate is the operand of a qualified expression whose type mark denotes a constrained array subtype.

8   (c) The aggregate is the expression of the component association of an enclosing (array or record) aggregate. Moreover, if this enclosing aggregate is a multidimensional array aggregate then it is itself in one of these three contexts.[7]

9   The bounds of an array aggregate that does not have an **others** choice are determined as follows. For an aggregate that has named associations, the bounds are determined by the smallest and largest choices given. For a positional aggregate, the lower bound is determined by the applicable index constraint if the aggregate appears in one of the contexts (a) through (c); otherwise, the lower bound is given by S$'$FIRST where S is the index subtype; in either case, the upper bound is determined by the number of components.

10   The evaluation of an array aggregate that is not a subaggregate proceeds in two steps. First, the choices of this aggregate and of its subaggregates, if any, are evaluated in some order that is not defined by the language. Second, the expressions of the component associations of the array aggregate are evaluated in some order that is not defined by the language; the expression of a named association is evaluated once for each associated component. The evaluation of a subaggregate consists of this second step (the first step is omitted since the choices have already been evaluated).

---

[7] See also Appendix G, AI-00177.

11   For the evaluation of an aggregate that is not a null array, a check is made that the index values defined by choices belong to the corresponding index subtypes, and also that the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent. For an n-dimensional multidimensional aggregate, a check is made that all (n − 1)-dimensional subaggregates have the same bounds. The exception CONSTRAINT_ERROR is raised if any of these checks fails.[8]

Note:

12   The allowed contexts for an array aggregate including an **others** choice are such that the bounds of such an aggregate are always known from the context.

13   **Examples of array aggregates with positional associations:**

```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
TABLE'(5, 8, 4, 1, others => 0) --  see 3.6
```

14   **Examples of array aggregates with named associations:**

```
(1 .. 5 => (1 .. 8 => 0.0))     -- two-dimensional
(1 .. N => new CELL)            -- N new cells,
                                -- in particular for N = 0

TABLE'(2 | 4 | 10 => 1, others => 0)
SCHEDULE'(MON .. FRI => TRUE,  others => FALSE)  --  see 3.6
SCHEDULE'(WED | SUN  => FALSE, others => TRUE)
```

15   **Examples of two-dimensional array aggregates:**

```
-- Three aggregates for the same value of type MATRIX (see 3.6):

((1.1, 1.2, 1.3), (2.1, 2.2, 2.3))
(1 => (1.1, 1.2, 1.3), 2 => (2.1, 2.2, 2.3))
(1 => (1 => 1.1, 2 => 1.2, 3 => 1.3),
     2 => (1 => 2.1, 2 => 2.2, 3 => 2.3))
```

16   **Examples of aggregates as initial values:**

```
A : TABLE := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0); -- A(1)=7, A(10)=0
B : TABLE := TABLE'(2 | 4 | 10 => 1,
                    others => 0);             -- B(1)=0, B(10)=1
C : constant MATRIX :=
      (1 .. 5 => (1 .. 8 => 0.0));            -- C'FIRST(1)=1,
                                              -- C'LAST(2)=8

D : BIT_VECTOR(M .. N) := (M .. N => TRUE);  -- see 3.6
E : BIT_VECTOR(M .. N) := (others => TRUE);
F : STRING(1 .. 1) := (1 => 'F');  -- a one component aggregate:
                                   -- same as "F"
```

---

[8]   See also Appendix G, AI-00018, AI-00019, AI-00265, and AI-00313.

17   **References:** actual parameter 6.4.1, aggregate 4.3, array type 3.6, assignment
compound delimiter 5.2, choice 3.7.3, component 3.3, component association 4.3,
component type 3.3, constrained array subtype 3.6, constraint 3.3, constraint_error
exception 11.1, dimension 3.6, evaluate 4.5, expression 4.4, formal parameter 6.1,
function 6.5, in some order 1.6, index constraint 3.6.1, index range 3.6, index subtype
3.6, index type 3.6, named component association 4.3, null array 3.6.1, object 3.2,
positional component association 4.3, qualified expression 4.7, raising of exceptions 11,
static expression 4.9, subcomponent 3.3, type 3.3

4.4 Expressions

1   An expression is a formula that defines the computation of a value.

```
2       expression ::=
            relation {and relation} | relation {and then relation}
          | relation {or relation}  | relation {or else relation}
          | relation {xor relation}

        relation ::=
            simple_expression [relational_operator simple_expression]
          | simple_expression [not] in range
          | simple_expression [not] in type_mark

        simple_expression ::=
           [unary_adding_operator] term {binary_adding_operator term}

        term ::= factor {multiplying_operator factor}

        factor ::= primary [** primary] | abs primary | not primary

        primary ::=
            numeric_literal | null | aggregate | string_literal
          | name | allocator | function_call | type_conversion
          | qualified_expression | (expression)
```

3   Each primary has a value and a type. The only names allowed as primaries are
named numbers; attributes that yield values; and names denoting objects (the
value of such a primary is the value of the object) or denoting values. Names
that denote formal parameters of mode **out** are not allowed as primaries;
names of their subcomponents are only allowed in the case of discriminants.

4   The type of an expression depends only on the type of its constituents and
on the operators applied; for an overloaded constituent or operator, the
determination of the constituent type, or the identification of the appropriate
operator, depends on the context. For each predefined operator, the operand
and result types are given in section 4.5.

⁵ **Examples of primaries:**
```
4.0                 -- real literal
PI                  -- named number
(1 .. 10 => 0)      -- array aggregate
SUM                 -- variable
INTEGER'LAST        -- attribute
SINE(X)             -- function call
COLOR'(BLUE)        -- qualified expression
REAL(M*N)           -- conversion
(LINE_COUNT + 10)   -- parenthesized expression
```

⁶ **Examples of expressions:**
```
VOLUME                      -- primary
not DESTROYED               -- factor
2*LINE_COUNT                -- term
-4.0                        -- simple expression
-4.0 + A                    -- simple expression
B**2 - 4.0*A*C              -- simple expression
PASSWORD(1 .. 3) = "BWV"    -- relation
COUNT in SMALL_INT          -- relation
COUNT not in SMALL_INT      -- relation
INDEX = 0 or ITEM_HIT       -- expression
(COLD and SUNNY) or WARM    -- expression
                            -- (parentheses are required)
A**(B**C)                   -- expression
                            -- (parentheses are required)
```

⁷ **References:** aggregate 4.3, allocator 4.8, array aggregate 4.3.2, attribute 4.1.4, binary adding operator 4.5 4.5.3, context of overload resolution 8.7, exponentiating operator 4.5 4.5.6, function call 6.4, multiplying operator 4.5 4.5.5, name 4.1, named number 3.2, null literal 3.8, numeric literal 2.4, object 3.2, operator 4.5, overloading 8.3, overloading an operator 6.7, qualified expression 4.7, range 3.5, real literal 2.4, relation 4.5.1, relational operator 4.5 4.5.2, result type 6.1, string literal 2.6, type 3.3, type conversion 4.6, type mark 3.3.2, unary adding operator 4.5 4.5.4, variable 3.2.1

4.5 Operators and Expression Evaluation

¹ The language defines the following six classes of operators. The corresponding operator symbols (except /=), and only those, can be used as designators in declarations of functions for user-defined operators. They are given in the order of increasing precedence.

²
```
logical_operator        ::= and | or  | xor
relational_operator     ::= =  | /= | <   | <= |> | >=
binary_adding_operator  ::= +  | -  | &
unary_adding_operator   ::= +  | -
multiplying_operator    ::= *  | /  | mod | rem
```

```
highest_precedence_operator  ::=  **  |  abs  |  not
```

3   The short-circuit control forms **and then** and **or else** have the same
    precedence as logical operators. The membership tests **in** and **not in** have the
    same precedence as relational operators.

4   For a term, simple expression, relation, or expression, operators of higher
    precedence are associated with their operands before operators of lower
    precedence. In this case, for a sequence of operators of the same precedence
    level, the operators are associated in textual order from left to right;
    parentheses can be used to impose specific associations.

5   The operands of a factor, of a term, of a simple expression, or of a relation, and
    the operands of an expression that does not contain a short-circuit control form,
    are evaluated in some order that is not defined by the language (but before
    application of the corresponding operator). The right operand of a short-circuit
    control form is evaluated if and only if the left operand has a certain value
    (see 4.5.1).

6   For each form of type declaration, certain of the above operators are *predefined*,
    that is, they are implicitly declared by the type declaration. For each such
    implicit operator declaration, the names of the parameters are LEFT and
    RIGHT for binary operators; the single parameter is called RIGHT for unary
    adding operators and for the unary operators **abs** and **not.** The effect of the
    predefined operators is explained in subsections 4.5.1 through 4.5.7.

7   The predefined operations on integer types either yield the mathematically
    correct result or raise the exception NUMERIC_ERROR. A predefined
    operation that delivers a result of an integer type (other than *universal_
    integer*) can only raise the exception NUMERIC_ERROR if the mathematical
    result is not a value of the type. The predefined operations on real types yield
    results whose accuracy is defined in section 4.5.7. A predefined operation that
    delivers a result of a real type (other than *universal_real*) can only raise the
    exception NUMERIC_ERROR if the result is not within the range of the safe
    numbers of the type, as explained in section 4.5.7.[9]

8   **Examples of precedence:**
```
not SUNNY or WARM    --  same as (not SUNNY) or WARM
X > 4.0 and Y > 0.0  --  same as (X > 4.0) and (Y > 0.0)

-4.0*A**2            --  same as -(4.0 * (A**2))
abs(1 + A) + B       --  same as (abs (1 + A)) + B
Y**(-3)              --  parentheses are necessary
A / B * C            --  same as (A/B)*C
A + (B + C)          --  evaluate B + C before adding it to A
```

---

[9]   See also Appendix G, AI-00387.

9   **References:** designator 6.1, expression 4.4, factor 4.4, implicit declaration 3.1, in some order 1.6, integer type 3.5.4, membership test 4.5.2, name 4.1, numeric_error exception 11.1, overloading 6.6 8.7, raising of an exception 11, range 3.5, real type 3.5.6, relation 4.4, safe number 3.5.6, short-circuit control form 4.5 4.5.1, simple expression 4.4, term 4.4, type 3.3, type declaration 3.3.1, universal_integer type 3.5.4, universal_real type 3.5.6

4.5.1 Logical Operators and Short-circuit Control Forms

1   The following logical operators are predefined for any boolean type and any one-dimensional array type whose components are of a boolean type; in either case the two operands have the same type.

2

Operator	Operation	Operand type	Result type
and	conjunction	any boolean type	same boolean type
		array of boolean components	same array type
or	inclusive disjunction	any boolean type	same boolean type
		array of boolean components	same array type
xor	exclusive disjunction	any boolean type	same boolean type
		array of boolean components	same array type

3   The operations on arrays are performed on a component-by-component basis on matching components, if any (as for equality, see 4.5.2). The bounds of the resulting array are those of the left operand. A check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. The exception CONSTRAINT_ERROR is raised if this check fails.[10]

4   The short-circuit control forms **and then** and **or else** are defined for two operands of a boolean type and deliver a result of the same type. The left operand of a short-circuit control form is always evaluated first. If the left operand of an expression with the control form **and then** evaluates to FALSE, the right operand is not evaluated and the value of the expression is FALSE. If the left operand of an expression with the control form **or else** evaluates to TRUE, the right operand is not evaluated and the value of the expression is

---

[10] See also Appendix G, AI-00426 and AI-00431.

TRUE. If both operands are evaluated, **and then** delivers the same result as **and**, and **or else** delivers the same result as **or.**

Note:

5   The conventional meaning of the logical operators is given by the following truth table:

6

A	B	A and B	A or B	A xor B
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

7   **Examples of logical operators:**
```
SUNNY or WARM
FILTER(1 .. 10) and FILTER(15 .. 24)   --   see 3.6.1
```

8   **Examples of short-circuit control forms:**
```
NEXT_CAR.OWNER /= null
    and then NEXT_CAR.OWNER.AGE > 25 -- see 3.8.1

N = 0 or else A(N) = HIT_VALUE
```

9   **References:** array type 3.6, boolean type 3.5.3, bound of an index range 3.6.1, component of an array 3.6, constraint_error exception 11.1, dimension 3.6, false boolean value 3.5.3, index subtype 3.6, matching components of arrays 4.5.2, null array 3.6.1, operation 3.3, operator 4.5, predefined operator 4.5, raising of exceptions 11, true boolean value 3.5.3, type 3.3

4.5.2 Relational Operators and Membership Tests

1   The equality and inequality operators are predefined for any type that is not limited. The other relational operators are the ordering operators < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). The ordering operators are predefined for any scalar type, and for any discrete array type, that is, a one-dimensional array type whose components are of a discrete type. The operands of each predefined relational operator have the same type. The result type is the predefined type BOOLEAN.

2   The relational operators have their conventional meaning: the result is equal to TRUE if the corresponding relation is satisfied; the result is FALSE otherwise. The inequality operator gives the complementary result to the equality operator: FALSE if equal, TRUE if not equal.

	Operator	Operation	Operand type	Result type
3	= /=	equality and inequality	any type	BOOLEAN
	< <= > >=	test for ordering	any scalar type	BOOLEAN
			discrete array type	BOOLEAN

4 Equality for the discrete types is equality of the values. For real operands whose values are *nearly* equal, the results of the predefined relational operators are given in section 4.5.7. Two access values are equal either if they designate the same object, or if both are equal to the null value of the access type.

5 For two array values or two record values of the same type, the left operand is equal to the right operand if and only if for each component of the left operand there is a *matching component* of the right operand and vice versa; and the values of matching components are equal, as given by the predefined equality operator for the component type. In particular, two null arrays of the same type are always equal; two null records of the same type are always equal.

6 For comparing two records of the same type, *matching components* are those which have the same component identifier.

7 For comparing two one-dimensional arrays of the same type, *matching components* are those (if any) whose index values match in the following sense: the lower bounds of the index ranges are defined to match, and the successors of matching indices are defined to match. For comparing two multidimensional arrays, matching components are those whose index values match in successive index positions.

8 If equality is explicitly defined for a limited type, it does not extend to composite types having subcomponents of the limited type (explicit definition of equality is allowed for such composite types).

9 The ordering operators <, <=, >, and >= that are defined for discrete array types correspond to *lexicographic* order using the predefined order relation of the component type. A null array is lexicographically less than any array having at least one component. In the case of nonnull arrays, the left operand is lexicographically less than the right operand if the first component of the left operand is less than that of the right; otherwise the left operand is lexicographically less than the right operand only if their first components are equal and the tail of the left operand is lexicographically less than that of the right (the tail consists of the remaining components beyond the first and can be null).

¹⁰ The membership tests **in** and **not in** are predefined for all types. The result type is the predefined type BOOLEAN. For a membership test with a range, the simple expression and the bounds of the range must be of the same scalar type; for a membership test with a type mark, the type of the simple expression must be the base type of the type mark. The evaluation of the membership test **in** yields the result TRUE if the value of the simple expression is within the given range, or if this value belongs to the subtype denoted by the given type mark; otherwise this evaluation yields the result FALSE (for a value of a real type, see 4.5.7). The membership test **not in** gives the complementary result to the membership test **in.**

¹¹ **Examples:**

```
X /= Y

"" < "A" and "A" < "AA"     --  TRUE
"AA" < "B" and "A" < "A "   --  TRUE

MY_CAR = null               -- true if MY_CAR has been set
                            -- to null (see 3.8.1)

MY_CAR = YOUR_CAR           -- true if we both share
                            -- the same car

MY_CAR._ALL = YOUR_CAR._ALL -- true if the two cars
                            -- are identical

N not in 1 .. 10     -- range membership test
TODAY in MON .. FRI  -- range membership test
TODAY in WEEKDAY     -- subtype membership test (see 3.5.1)
ARCHIVE in DISK_UNIT -- subtype membership test (see 3.7.3)
```

Notes:

¹² No exception is ever raised by a predefined relational operator or by a membership test, but an exception can be raised by the evaluation of the operands.

¹³ If a record type has components that depend on discriminants, two values of this type have matching components if and only if their discriminants are equal. Two nonnull arrays have matching components if and only if the value of the attribute LENGTH(N) for each index position N is the same for both.

¹⁴ **References:** access value 3.8, array type 3.6, base type 3.3, belong to a subtype 3.3, boolean predefined type 3.5.3, bound of a range 3.5, component 3.3, component identifier 3.7, component type 3.3, composite type 3.3, designate 3.8, dimension 3.6, discrete type 3.5, evaluation 4.5, exception 11, index 3.6, index range 3.6, limited type 7.4.4, null access value 3.8, null array 3.6.1, null record 3.7, object 3.2.1, operation 3.3, operator 4.5, predefined operator 4.5, raising of exceptions 11, range 3.5, record type 3.7, scalar type 3.5, simple expression 4.4, subcomponent 3.3, successor 3.5.5, type 3.3, type mark 3.3.2

4.5.3 Binary Adding Operators

1   The binary adding operators + and – are predefined for any numeric type and have their conventional meaning. The catenation operators & are predefined for any one-dimensional array type that is not limited.

2

Operator	Operation	Left operand type	Right operand type	Result type
+	addition	any numeric type	same numeric type	same numeric type
–	subtraction	any numeric type	same numeric type	same numeric type
&	catenation	any array type	same array type	same array type
		any array type	the component type	same array type
		the component type	any array type	same array type
		the component type	the component type	any array type

3   For real types, the accuracy of the result is determined by the operand type (see 4.5.7).

4   If both operands are one-dimensional arrays, the result of the catenation is a one-dimensional array whose length is the sum of the lengths of its operands, and whose components comprise the components of the left operand followed by the components of the right operand. The lower bound of this result is the lower bound of the left operand, unless the left operand is a null array, in which case the result of the catenation is the right operand.

5   If either operand is of the component type of an array type, the result of the catenation is given by the above rules, using in place of this operand an array having this operand as its only component and having the lower bound of the index subtype of the array type as its lower bound.

6   The exception CONSTRAINT_ERROR is raised by catenation if the upper bound of the result exceeds the range of the index subtype, unless the result is a null array. This exception is also raised if any operand is of the component type but has a value that does not belong to the component subtype.

Examples:

```
Z + 0.1      --  Z must be of a real type

"A" & "BCD"  --  catenation of two string literals

'A' & "BCD"  --  catenation of a character literal
             --  and a string literal

'A' & 'A'    --  catenation of two character literals
```

8 **References:** array type 3.6, character literal 2.5, component type 3.3, constraint_ error exception 11.1, dimension 3.6, index subtype 3.6, length of an array 3.6.2, limited type 7.4.4, null array 3.6.1, numeric type 3.5, operation 3.3, operator 4.5, predefined operator 4.5, raising of exceptions 11, range of an index subtype 3.6.1, real type 3.5.6, string literal 2.6, type 3.3

4.5.4 Unary Adding Operators

1 The unary adding operators + and – are predefined for any numeric type and have their conventional meaning. For each of these operators, the operand and the result have the same type.

2

Operator	Operation	Operand type	Result type
+	identity	any numeric type	same numeric type
–	negation	any numeric type	same numeric type

3 **References:** numeric type 3.5, operation 3.3, operator 4.5, predefined operator 4.5, type 3.3

4.5.5 Multiplying Operators

1 The operators * and / are predefined for any integer and any floating point type and have their conventional meaning; the operators **mod** and **rem** are predefined for any integer type. For each of these operators, the operands and the result have the same base type. For floating point types, the accuracy of the result is determined by the operand type (see 4.5.7).

2

Operator	Operation	Operand type	Result type
*	multiplication	any integer type	same integer type
		any floating point type	same floating point type
/	integer division	any integer type	same integer type

2	Operator	Operation	Operand type	Result type
		floating division	any floating point type	same floating point type
	mod	modulus	any integer type	same integer type
	rem	remainder	any integer type	same integer type

3   Integer division and remainder are defined by the relation

```
A = (A/B)*B + (A rem B)
```

4   where (A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Integer division satisfies the identity

```
(-A)/B = -(A/B) = A/(-B)
```

5   The result of the modulus operation is such that (A **mod** B) has the sign of B and an absolute value less than the absolute value of B; in addition, for some integer value N, this result must satisfy the relation

```
A = B*N + (A mod B)
```

6   For each fixed point type, the following multiplication and division operators, with an operand of the predefined type INTEGER, are predefined.

7	Operator	Operation	Left operand type	Right operand type	Result type
	*	multiplication	any fixed point type	INTEGER	same as left
			INTEGER	any fixed point type	same as right
	/	division	any fixed point type	INTEGER	same as left

8   Integer multiplication of fixed point values is equivalent to repeated addition. Division of a fixed point value by an integer does not involve a change in type but is approximate (see 4.5.7).[11]

---

[11] See also Appendix G, AI-00475.

9   Finally, the following multiplication and division operators are declared in the predefined package STANDARD. These two special operators apply to operands of all fixed point types (it is a consequence of other rules that they cannot be renamed or given as generic actual parameters).

10

Operator	Operation	Left operand type[1]	Right operand type[1]	Result type
*	multiplication	any fixed point type	any fixed point type	*universal_fixed*
/	division	any fixed point type	any fixed point type	*universal_fixed*

[1]See also Appendix G, AI-00020 and AI-00376.

11   Multiplication of operands of the same or of different fixed point types is exact and delivers a result of the anonymous predefined fixed point type *universal_fixed* whose delta is arbitrarily small. The result of any such multiplication must always be explicitly converted to some numeric type. This ensures explicit control of the accuracy of the computation. The same considerations apply to division of a fixed point value by another fixed point value. No other operators are defined for the type *universal_fixed*.[12]

12   The exception NUMERIC_ERROR is raised by integer division, **rem**, and **mod** if the right operand is zero.[13]

13   **Examples:**
```
I : INTEGER := 1;
J : INTEGER := 2;
K : INTEGER := 3;

X : REAL digits 6 := 1.0;                --    see 3.5.7
Y : REAL digits 6 := 2.0;
```

---

[12] See also Appendix G, AI-00235.
[13] See also Appendix G, AI-00387.

```
F : FRACTION delta 0.0001 := 0.1;      --    see 3.5.9
G : FRACTION delta 0.0001 := 0.1;
```

Expression	Value	Result type
I*J	2	same as I and J, that is, INTEGER
K/J	1	same as K and J, that is, INTEGER
K **mod** J	1	same as K and J, that is, INTEGER
X/Y	0.5	same as X and Y, that is, REAL
F/2	0.05	same as F, that is, FRACTION
3*F	0.3	same as F, that is, FRACTION
F*G	0.01	*universal_fixed*, conversion needed
FRACTION(F*G)	0.01	FRACTION, as stated by the conversion
REAL(J)*Y	4.0	REAL, the type of both operands after conversion of J

Notes:

14 For positive A and B, A/B is the quotient and A **rem** B is the remainder when A is divided by B. The following relations are satisfied by the **rem** operator:

```
A     rem (-B) =   A rem B
(-A) rem    B  = -(A rem B)
```

15 For any integer K, the following identity holds:

```
A   mod   B  =  (A + K*B) mod B
```

16 The relations between integer division, remainder, and modulus are illustrated by the following table:

A	B	A/B	A rem B	A mod B
10	5	2	0	0
11	5	2	1	1
12	5	2	2	2
13	5	2	3	3
14	5	2	4	4
10	−5	−2	0	0
11	−5	−2	1	−4
12	−5	−2	2	−3

A	B	A/B	A rem B	A mod B
13	−5	−2	3	−2
14	−5	−2	4	−1
−10	5	−2	0	0
−11	5	−2	−1	4
−12	5	−2	−2	3
−13	5	−2	−3	2
−14	5	−2	−4	1
−10	−5	2	0	0
−11	−5	2	−1	−1
−12	−5	2	−2	−2
−13	−5	2	−3	−3
−14	−5	2	−4	−4

17 **References:** actual parameter 6.4.1, base type 3.3, declaration 3.1, delta of a fixed point type 3.5.9, fixed point type 3.5.9, floating point type 3.5.7, generic formal subprogram 12.1, integer type 3.5.4, numeric type 3.5, numeric_error exception 11.1, predefined operator 4.5, raising of exceptions 11, renaming declaration 8.5, standard predefined package 8.6, type conversion 4.6

4.5.6 Highest Precedence Operators

1 The highest precedence unary operator **abs** is predefined for any numeric type. The highest precedence unary operator **not** is predefined for any boolean type and any one-dimensional array type whose components have a boolean type.

2

Operator	Operation	Operand type	Result type
abs	absolute value	any numeric type	same numeric type
not	logical negation	any boolean type	same boolean type
		array of boolean components	same array type

3 The operator **not** that applies to a one-dimensional array of boolean components yields a one-dimensional boolean array with the same bounds; each component of the result is obtained by logical negation of the corresponding component of the operand (that is, the component that has the same index value).

4 The highest precedence *exponentiating* operator ** is predefined for each integer type and for each floating point type. In either case the right operand, called the exponent, is of the predefined type INTEGER.

5

Operator	Operation	Left operand type	Right operand type	Result type
**	exponentiation	any integer type	INTEGER	same as left
		any floating point type	INTEGER	same as left

6 Exponentiation with a positive exponent is equivalent to repeated multiplication of the left operand by itself, as indicated by the exponent and from left to right. For an operand of a floating point type, the exponent can be negative, in which case the value is the reciprocal of the value with the positive exponent. Exponentiation by a zero exponent delivers the value one. Exponentiation of a value of a floating point type is approximate (see 4.5.7). Exponentiation of an integer raises the exception CONSTRAINT_ERROR for a negative exponent.[14]

7 **References:** array type 3.6, boolean type 3.5.3, bound of an array 3.6.1, component of an array 3.6, constraint_error exception 11.1, dimensionality 3.6, floating point type 3.5.9, index 3.6, integer type 3.5.4, multiplication operation 4.5.5, predefined operator 4.5, raising of exceptions 11

4.5.7 Accuracy of Operations with Real Operands

1 A real subtype specifies a set of model numbers.[15] Both the accuracy required from any basic or predefined operation giving a real result, and the result of any predefined relation between real operands are defined in terms of these model numbers.

2 A *model interval* of a subtype is any interval whose bounds are model numbers of the subtype. The model interval associated with a value that belongs to a real subtype is the smallest model interval (of the subtype) that includes the value. (The model interval associated with a model number of a subtype consists of that number only.)

3 For any basic operation or predefined operator that yields a result of a real subtype, the required bounds on the result are given by a model interval defined as follows:

---

[14] See also Appendix G, AI-00137.
[15] See also Appendix G, AI-00407.

4   • The result model interval is the smallest model interval (of the result subtype) that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation, when each operand is given any value of the model interval (of the operand subtype) defined for the operand.[16]

5   • The model interval of an operand that is itself the result of an operation, other than an implicit conversion, is the result model interval of this operation.

6   • The model interval of an operand whose value is obtained by implicit conversion of a universal expression is the model interval associated with this value within the operand subtype.

7   The result model interval is undefined if the absolute value of one of the above mathematical results exceeds the largest safe number of the result type. Whenever the result model interval is undefined, it is highly desirable that the exception NUMERIC_ERROR be raised if the implementation cannot produce an actual result that is in the range of safe numbers. This is, however, not required by the language rules, in recognition of the fact that certain target machines do not permit easy detection of overflow situations. The value of the attribute MACHINE_OVERFLOWS indicates whether the target machine raises the exception NUMERIC_ERROR in overflow situations (see 13.7.3).[17]

8   The safe numbers of a real type are defined (see 3.5.6) as a superset of the model numbers, for which error bounds follow the same rules as for model numbers. Any definition given in this section in terms of model intervals can therefore be extended to safe intervals of safe numbers. A consequence of this extension is that an implementation is not allowed to raise the exception NUMERIC_ERROR when the result interval is a safe interval.

9   For the result of exponentiation, the model interval defining the bounds on the result is obtained by applying the above rules to the sequence of multiplications defined by the exponent, and to the final division in the case of a negative exponent.

10   For the result of a relation between two real operands, consider for each operand the model interval (of the operand subtype) defined for the operand; the result can be any value obtained by applying the mathematical comparison to values arbitrarily chosen in the corresponding operand model intervals. If either or both of the operand model intervals is undefined (and if neither of the operand evaluations raises an exception) then the result of the comparison is allowed to be any possible value (that is, either TRUE or FALSE).

---

[16] See also Appendix G, AI-00516.
[17] See also Appendix G, AI-00387.

₁₁ The result of a membership test is defined in terms of comparisons of the operand value with the lower and upper bounds of the given range or type mark (the usual rules apply to these comparisons).

Note:

₁₂ For a floating point type the numbers 15.0, 3.0, and 5.0 are always model numbers. Hence X/Y where X equals 15.0 and Y equals 3.0 yields exactly 5.0 according to the above rules. In the general case, division does not yield model numbers and in consequence one cannot assume that $(1.0/X)*X = 1.0$.

₁₃ **References:** attribute 4.1.4, basic operation 3.3.3, bound of a range 3.5, error bound 3.5.6, exponentiation operation 4.5.6, false boolean value 3.5.3, floating point type 3.5.9, machine_overflows attribute 13.7.1, membership test 4.5.2, model number 3.5.6, multiplication operation 4.5.5, numeric_error exception 11.1, predefined operation 3.3.3, raising of exceptions 11, range 3.5, real type 3.5.6, relation 4.4, relational operator 4.5.2 4.5, safe number 3.5.6, subtype 3.3, true boolean value 3.5.3, type conversion 4.6, type mark 3.3.2, universal expression 4.10

4.6 Type Conversions

₁ The evaluation of an explicit type conversion evaluates the expression given as the operand, and converts the resulting value to a specified *target* type. Explicit type conversions are allowed between closely related types as defined below.

₂
```
    type_conversion ::= type_mark(expression)
```

₃ The target type of a type conversion is the base type of the type mark. The type of the operand of a type conversion must be determinable independently of the context (in particular, independently of the target type). Furthermore, the operand of a type conversion is not allowed to be a literal **null**, an allocator, an aggregate, or a string literal; an expression enclosed by parentheses is allowed as the operand of a type conversion only if the expression alone is allowed.

₄ A conversion to a subtype consists of a conversion to the target type followed by a check that the result of the conversion belongs to the subtype. A conversion of an operand of a given type to the type itself is allowed.

₅ The other allowed explicit type conversions correspond to the following three cases:

₆   (a) Numeric types

₇   The operand can be of any numeric type; the value of the operand is converted to the target type which must also be a numeric type. For conversions involving real types, the result is within the accuracy of the specified subtype (see 4.5.7). The conversion of a real value to an

integer type rounds to the nearest integer; if the operand is halfway between two integers (within the accuracy of the real subtype) rounding may be either up or down.

8   (b)   Derived types

9   The conversion is allowed if one of the target type and the operand type is derived from the other, directly or indirectly, or if there exists a third type from which both types are derived, directly or indirectly.

10   (c)   Array types

11   The conversion is allowed if the operand type and the target type are array types that satisfy the following conditions: both types must have the same dimensionality; for each index position the index types must either be the same or be convertible to each other; the component types must be the same; finally, if the component type is a type with discriminants or an access type, the component subtypes must be either both constrained or both unconstrained. If the type mark denotes an unconstrained array type, then, for each index position, the bounds of the result are obtained by converting the bounds of the operand to the corresponding index type of the target type. If the type mark denotes a constrained array subtype, then the bounds of the result are those imposed by the type mark. In either case, the value of each component of the result is that of the matching component of the operand (see 4.5.2).

12   In the case of conversions of numeric types and derived types, the exception CONSTRAINT_ERROR is raised by the evaluation of a type conversion if the result of the conversion fails to satisfy a constraint imposed by the type mark.

13   In the case of array types, a check is made that any constraint on the component subtype is the same for the operand array type as for the target array type. If the type mark denotes an unconstrained array type and if the operand is not a null array, then, for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type. If the type mark denotes a constrained array subtype, a check is made that for each component of the operand there is a matching component of the target subtype, and vice versa. The exception CONSTRAINT_ERROR is raised if any of these checks fails.[18]

14   If a conversion is allowed from one type to another, the reverse conversion is also allowed. This reverse conversion is used where an actual parameter of mode **in out** or **out** has the form of a type conversion of a (variable) name as explained in section 6.4.1.

---

[18] See also Appendix G, AI-00313.

15   Apart from the explicit type conversions, the only allowed form of type
      conversion is the implicit conversion of a value of the type *universal_integer*
      or *universal_real* into another numeric type. An implicit conversion of an
      operand of type *universal_integer* to another integer type, or of an operand of
      type *universal_real* to another real type, can only be applied if the operand is
      either a numeric literal, a named number, or an attribute; such an operand is
      called a *convertible* universal operand in this section. An implicit conversion of
      a convertible universal operand is applied if and only if the innermost complete
      context (see 8.7) determines a unique (numeric) target type for the implicit
      conversion, and there is no legal interpretation of this context without this
      conversion.

Notes:

15   The rules for implicit conversions imply that no implicit conversion is ever
      applied to the operand of an explicit type conversion. Similarly, implicit
      conversions are not applied if both operands of a predefined relational operator
      are convertible universal operands.

16   The language allows implicit subtype conversions in the case of array types
      (see 5.2.1). An explicit type conversion can have the effect of a change of
      representation (in particular see 13.6). Explicit conversions are also used for
      actual parameters (see 6.4).

17   **Examples of numeric type conversion:**

```
REAL(2*J)      -- value is converted to floating point
INTEGER(1.6)   -- value is 2
INTEGER(-0.4)  -- value is 0
```

18   **Example of conversion between derived types:**

```
type A_FORM is new B_FORM;

X : A_FORM;
Y : B_FORM;

X := A_FORM(Y);
Y := B_FORM(X);  --  the reverse conversion
```

19   **Examples of conversions between array types:**

```
type SEQUENCE is array (INTEGER range <>) of INTEGER;
subtype DOZEN is SEQUENCE(1 .. 12);
LEDGER : array(1 .. 100) of INTEGER;

SEQUENCE(LEDGER)            --  bounds are those of LEDGER
SEQUENCE(LEDGER(31 .. 42))  --  bounds are 31 and 42
DOZEN(LEDGER(31 .. 42))     --  bounds are those of DOZEN
```

20 **Examples of implicit conversions:**

```
X : INTEGER := 2;

X + 1 + 2               -- implicit conversion of
                        -- each integer literal

1 + 2 + X               -- implicit conversion of
                        -- each integer literal

X + (1 + 2)             -- implicit conversion of
                        -- each integer literal

2 = (1 + 1)             -- no implicit conversion:
                        -- the type is universal_integer

A'LENGTH = B'LENGTH     -- no implicit conversion:
                        -- the type is universal_integer

C : constant := 3 + 2;  -- no implicit conversion:
                        -- the type is universal_integer

X = 3 and 1 = 2         -- implicit conversion of 3,
                        -- but not of 1 and 2
```

21 **References:** actual parameter 6.4.1, array type 3.6, attribute 4.1.4, base type 3.3, belong to a subtype 3.3, component 3.3, constrained array subtype 3.6, constraint_ error exception 11.1, derived type 3.4, dimension 3.6, expression 4.4, floating point type 3.5.7, index 3.6, index subtype 3.6, index type 3.6, integer type 3.5.4, matching component 4.5.2, mode 6.1, name 4.1, named number 3.2, null array 3.6.1, numeric literal 2.4, numeric type 3.5, raising of exceptions 11, real type 3.5.6, representation 13.1, statement 5, subtype 3.3, type 3.3, type mark 3.3.2, unconstrained array type 3.6, universal_integer type 3.5.4, universal_real type 3.5.6, variable 3.2.1

4.7 Qualified Expressions

1 A qualified expression is used to state explicitly the type, and possibly the subtype, of an operand that is the given expression or aggregate.

2
```
qualified_expression ::=
    type_mark'(expression) | type_mark'aggregate
```

3 The operand must have the same type as the base type of the type mark. The value of a qualified expression is the value of the operand. The evaluation of a qualified expression evaluates the operand and checks that its value belongs to the subtype denoted by the type mark. The exception CONSTRAINT_ERROR is raised if this check fails.

⁴ **Examples:**

```
type MASK is (FIX, DEC, EXP, SIGNIF);
type CODE is (FIX, CLA, DEC, TNZ, SUB);

PRINT (MASK'(DEC)); --  DEC is of type MASK
PRINT (CODE'(DEC)); --  DEC is of type CODE

for J in CODE'(FIX) .. CODE'(DEC) loop ... -- qualification
                                           -- needed for
                                           -- either FIX
                                           -- or DEC

for J in CODE range FIX .. DEC loop ...    -- qualification
                                           -- unnecessary

for J in CODE'(FIX) .. DEC loop ...        -- qualification
                                           -- unnecessary
                                           -- for DEC

DOZEN'(1 | 3 | 5 | 7 => 2, others => 0)    -- see 4.6
```

Notes:

⁵ Whenever the type of an enumeration literal or aggregate is not known from the context, a qualified expression can be used to state the type explicitly. For example, an overloaded enumeration literal must be qualified in the following cases: when given as a parameter in a subprogram call to an overloaded subprogram that cannot otherwise be identified on the basis of remaining parameter or result types, in a relational expression where both operands are overloaded enumeration literals, or in an array or loop parameter range where both bounds are overloaded enumeration literals. Explicit qualification is also used to specify which one of a set of overloaded parameterless functions is meant, or to constrain a value to a given subtype.

⁶ **References:** aggregate 4.3, array 3.6, base type 3.3, bound of a range 3.5, constraint_ error exception 11.1, context of overload resolution 8.7, enumeration literal 3.5.1, expression 4.4, function 6.5, loop parameter 5.5, overloading 8.5, raising of exceptions 11, range 3.3, relation 4.4, subprogram 6, subprogram call 6.4, subtype 3.3, type 3.3, type mark 3.3.2

4.8 Allocators

¹ The evaluation of an allocator creates an object and yields an access value that designates the object.

²
```
allocator ::=
    new subtype_indication | new qualified_expression
```

3   The type of the object created by an allocator is the base type of the type mark given in either the subtype indication or the qualified expression. For an allocator with a qualified expression, this expression defines the initial value of the created object. The type of the access value returned by an allocator must be determinable solely from the context, but using the fact that the value returned is of an access type having the named designated type.

4   The only allowed forms of constraint in the subtype indication of an allocator are index and discriminant constraints. If an allocator includes a subtype indication and if the type of the object created is an array type or a type with discriminants that do not have default expressions, then the subtype indication must either denote a constrained subtype, or include an explicit index or discriminant constraint.

5   If the type of the created object is an array type or a type with discriminants, then the created object is always constrained. If the allocator includes a subtype indication, the created object is constrained either by the subtype or by the default discriminant values. If the allocator includes a qualified expression, the created object is constrained by the bounds or discriminants of the initial value. For other types, the subtype of the created object is the subtype defined by the subtype indication of the access type definition.[19]

6   For the evaluation of an allocator, the elaboration of the subtype indication or the evaluation of the qualified expression is performed first. The new object is then created. Initializations are then performed as for a declared object (see 3.2.1); the initialization is considered explicit in the case of a qualified expression; any initializations are implicit in the case of a subtype indication. Finally, an access value that designates the created object is returned.

7   An implementation must guarantee that any object created by the evaluation of an allocator remains allocated for as long as this object or one of its subcomponents is accessible directly or indirectly, that is, as long as it can be denoted by some name. Moreover, if an object or one of its subcomponents belongs to a task type, it is considered to be accessible as long as the task is not terminated. An implementation may (but need not) reclaim the storage occupied by an object created by an allocator, once this object has become inaccessible.[20]

In DEC Ada, storage is reclaimed only upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration. In other words, storage for an inaccessible object of an access type is not reclaimed until the collection allocated for the access type is reclaimed

---

[19] See also Appendix G, AI-00150, AI-00331, and AI-00397.
[20] See also Appendix G, AI-00356.

(see also 13.2). For more detailed information on DEC Ada storage allocation and deallocation, see the DEC Ada run-time reference manuals.

8   When an application needs closer control over storage allocation for objects designated by values of an access type, such control may be achieved by one or more of the following means:

9   (a)   The total amount of storage available for the collection of objects of an access type can be set by means of a length clause (see 13.2).

10   (b)   The pragma CONTROLLED informs the implementation that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program.

```
pragma CONTROLLED (access_type_simple_name);
```

11   A pragma CONTROLLED for a given access type is allowed at the same places as a representation clause for the type (see 13.1). This pragma is not allowed for a derived type.[21]

12   (c)   The explicit deallocation of the object designated by an access value can be achieved by calling a procedure obtained by instantiation of the predefined generic library procedure UNCHECKED_DEALLOCATION (see 13.10.1).

13 The exception STORAGE_ERROR is raised by an allocator if there is not enough storage. Note also that the exception CONSTRAINT_ERROR can be raised by the evaluation of the qualified expression, by the elaboration of the subtype indication, or by the initialization.[22]

14 **Examples for access types declared in section 3.8:**

```
new CELL'(0, null, null) -- initialized explicitly

new CELL'(VALUE => 0,
          SUCC => null,
          PRED => null)  -- initialized explicitly

new CELL                 -- not initialized

new MATRIX(1 .. 10, 1 .. 20)             -- the bounds only
                                         -- are given

new MATRIX'(1 .. 10 => (1 .. 20 => 0.0)) -- initialized
                                         -- explicitly
```

---

[21] See also Appendix G, AI-00294.
[22] See also Appendix G, AI-00397.

```
new BUFFER(100)                                    -- the discriminant
                                                   -- only is given
new BUFFER'(SIZE => 80,
            POS => 0,
            VALUE => (1 .. 80 => 'A'))   -- initialized
                                         -- explicitly
```

15 **References:** access type 3.8, access type definition 3.8, access value 3.8, array type 3.6, block statement 5.6, bound of an array 3.6.1, collection 3.8, constrained subtype 3.3, constraint 3.3, constraint_error exception 11.1, context of overload resolution 8.7, derived type 3.4, designate 3.8, discriminant 3.3, discriminant constraint 3.7.2, elaboration 3.9, evaluation of a qualified expression 4.7, generic procedure 12.1, index constraint 3.6.1, initial value 3.2.1, initialization 3.2.1, instantiation 12.3, length clause 13.2, library unit 10.1, main program 10.1, name 4.1, object 3.2.1, object declaration 3.2.1, pragma 2.8, procedure 6, qualified expression 4.7, raising of exceptions 11, representation clause 13.1, simple name 4.1, storage_error exception 11.1, subcomponent 3.3, subprogram body 6.3, subtype 3.3, subtype indication 3.3.2, task body 9.1, task type 9.2, terminated task 9.4, type 3.3, type declaration 3.3.1, type mark 3.3.2 type with discriminants 3.3

4.9 Static Expressions and Static Subtypes

1 Certain expressions of a scalar type are said to be *static.* Similarly, certain discrete ranges are said to be static, and the type marks of certain scalar subtypes are said to denote static subtypes.

2 An expression of a scalar type is said to be static if and only if every primary is one of those listed in (a) through (h) below, every operator denotes a predefined operator, and the evaluation of the expression delivers a value (that is, it does not raise an exception):[23]

3   (a)   An enumeration literal (including a character literal).

4   (b)   A numeric literal.

5   (c)   A named number.

6   (d)   A constant explicitly declared by a constant declaration with a static subtype, and initialized with a static expression.[24]

7   (e)   A function call whose function name is an operator symbol that denotes a predefined operator, including a function name that is an expanded name; each actual parameter must also be a static expression.

---

[23] See also Appendix G, AI-00128, AI-00190, and AI-00219.

[24] See also Appendix G, AI-00001 and AI-00163.

8   (f)   A language-defined attribute of a static subtype; for an attribute that is a function, the actual parameter must also be a static expression.

9   (g)   A qualified expression whose type mark denotes a static subtype and whose operand is a static expression.

10   (h)   A static expression enclosed in parentheses.

11   A static range is a range whose bounds are static expressions. A static range constraint is a range constraint whose range is static. A static subtype is either a scalar base type, other than a generic formal type; or a scalar subtype formed by imposing on a static subtype either a static range constraint, or a floating or fixed point constraint whose range constraint, if any, is static. A static discrete range is either a static subtype or a static range. A static index constraint is an index constraint for which each index subtype of the corresponding array type is static, and in which each discrete range is static. A static discriminant constraint is a discriminant constraint for which the subtype of each discriminant is static, and in which each expression is static.[25]

Notes:

12   The accuracy of the evaluation of a static expression having a real type is defined by the rules given in section 4.5.7. If the result is not a model number (or a safe number) of the type, the value obtained by this evaluation at compilation time need not be the same as the value that would be obtained by an evaluation at run time.

13   Array attributes are not static: in particular, the RANGE attribute is not static.

14   **References:** actual parameter 6.4.1, attribute 4.1.4, base type 3.3, bound of a range 3.5, character literal 2.5, constant 3.2.1, constant declaration 3.2.1, discrete range 3.6, discrete type 3.5, enumeration literal 3.5.1, exception 11, expression 4.4, function 6.5, generic actual parameter 12.3, generic formal type 12.1.2, implicit declaration 3.1, initialize 3.2.1, model number 3.5.6, named number 3.2, numeric literal 2.4, predefined operator 4.5, qualified expression 4.7, raising of exceptions 11, range constraint 3.5, safe number 3.5.6, scalar type 3.5, subtype 3.3, type mark 3.3.2

---

[25] See also Appendix G, AI-00023, AI-00251, and AI-00409.

4.10 Universal Expressions

1   A *universal_expression* is either an expression that delivers a result of type *universal_integer* or one that delivers a result of type *universal_real.*

2   The same operations are predefined for the type *universal_integer* as for any integer type. The same operations are predefined for the type *universal_real* as for any floating point type. In addition, these operations include the following multiplication and division operators:

3

Operator	Operation	Left operand type	Right operand type	Result type
*	multiplication	*universal_real*	*universal_ integer*	*universal_real*
		universal_ integer	*universal_real*	*universal_real*
/	division	*universal_real*	*universal_ integer*	*universal_real*

4   The accuracy of the evaluation of a universal expression of type *universal_ real* is at least as good as that of the most accurate predefined floating point type supported by the implementation, apart from *universal_real* itself. Furthermore, if a universal expression is a static expression, then the evaluation must be exact.[26]

5   For the evaluation of an operation of a nonstatic universal expression, an implementation is allowed to raise the exception NUMERIC_ERROR only if the result of the operation is a real value whose absolute value exceeds the largest safe number of the most accurate predefined floating point type (excluding *universal_real*), or an integer value greater than SYSTEM.MAX_ INT or less than SYSTEM.MIN_INT.[27]

Note:

6   It is a consequence of the above rules that the type of a universal expression is *universal_integer* if every primary contained in the expression is of this type (excluding actual parameters of attributes that are functions, and excluding right operands of exponentiation operators) and that otherwise the type is *universal_real.*

---

[26] See also Appendix G, AI-00103, AI-00209, and AI-00405.
[27] See also Appendix G, AI-00181 and AI-00387.

7 **Examples:**
```
1 + 1       -- 2
abs(-10)*3  -- 30

KILO : constant := 1000;
MEGA : constant := KILO*KILO;    -- 1_000_000
LONG : constant := FLOAT'DIGITS*2;

HALF_PI    : constant := PI/2;              -- see 3.2.2
DEG_TO_RAD : constant := HALF_PI/90;
RAD_TO_DEG : constant := 1.0/DEG_TO_RAD;
    -- equivalent to
    -- 1.0/((3.14159_26536/2)/90)
```

8 **References:** actual parameter 6.4.1, attribute 4.1.4, evaluation of an expression 4.5, floating point type 3.5.9, function 6.5, integer type 3.5.4, multiplying operator 4.5 4.5.5, predefined operation 3.3.3, primary 4.4, real type 3.5.6, safe number 3.5.6, system.max_int 13.7, system.min_int 13.7, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6

5

Statements

1 A *statement* defines an action to be performed; the process by which a statement achieves its action is called *execution* of the statement.

2 This chapter describes the general rules applicable to all statements. Some specific statements are discussed in later chapters. Procedure call statements are described in chapter 6 on subprograms. Entry call, delay, accept, select, and abort statements are described in chapter 9 on tasks. Raise statements are described in chapter 11 on exceptions, and code statements in chapter 13. The remaining forms of statements are presented in this chapter.

3 **References:** abort statement 9.10, accept statement 9.5, code statement 13.8, delay statement 9.6, entry call statement 9.5, procedure call statement 6.4, raise statement 11.3, select statement 9.7

5.1 Simple and Compound Statements—Sequences of Statements

1 A statement is either simple or compound. A simple statement encloses no other statement. A compound statement can enclose simple statements and other compound statements.

```
2     sequence_of_statements ::= statement {statement}

      statement ::=
         {label} simple_statement | {label} compound_statement

      simple_statement ::= null_statement
          | assignment_statement | procedure_call_statement
          | exit_statement        | return_statement
          | goto_statement        | entry_call_statement
          | delay_statement       | abort_statement
          | raise_statement       | code_statement

      compound_statement ::=
            if_statement          | case_statement
          | loop_statement        | block_statement
          | accept_statement      | select_statement
```

```
label ::= <<label_simple_name>>

null_statement ::= null;
```

3   A statement is said to be *labeled* by the label name of any label of the statement. A label name, and similarly a loop or block name, is implicitly declared at the end of the declarative part of the innermost block statement, subprogram body, package body, task body, or generic body that encloses the labeled statement, the named loop statement, or the named block statement, as the case may be. For a block statement without a declarative part, an implicit declarative part (and preceding **declare**) is assumed.

4   The implicit declarations for different label names, loop names, and block names occur in the same order as the beginnings of the corresponding labeled statements, loop statements, and block statements. Distinct identifiers must be used for all label, loop, and block names that are implicitly declared within the body of a program unit, including within block statements enclosed by this body, but excluding within other enclosed program units (a program unit is either a subprogram, a package, a task unit, or a generic unit).

5   Execution of a null statement has no other effect than to pass to the next action.

6   The execution of a sequence of statements consists of the execution of the individual statements in succession until the sequence is completed, or a transfer of control takes place. A transfer of control is caused either by the execution of an exit, return, or goto statement; by the selection of a terminate alternative; by the raising of an exception; or (indirectly) by the execution of an abort statement.

7   **Examples of labeled statements:**
```
<<HERE>> <<ICI>> <<AQUI>> <<HIER>> null;

<<AFTER>> X := 1;
```

Note:

8   The scope of a declaration starts at the place of the declaration itself (see 8.2). In the case of a label, loop, or block name, it follows from this rule that the scope of the *implicit* declaration starts before the first *explicit* occurrence of the corresponding name, since this occurrence is either in a statement label, a loop statement, a block statement, or a goto statement. An implicit declaration in a block statement may hide a declaration given in an outer program unit or block statement (according to the usual rules of hiding explained in section 8.3).

9 **References:** abort statement 9.10, accept statement 9.5, assignment statement 5.2, block name 5.6, block statement 5.6, case statement 5.4, code statement 13.8, declaration 3.1, declarative part 3.9, delay statement 9.6, entry call statement 9.5, exception 11, exit statement 5.7, generic body 12.1, generic unit 12, goto statement 5.9, hiding 8.3, identifier 2.3, if statement 5.3, implicit declaration 3.1, loop name 5.5, loop statement 5.5, package 7, package body 7.1, procedure call statement 6.4, program unit 6, raise statement 11.3, raising of exceptions 11, return statement 5.8, scope 8.2, select statement 9.7, simple name 4.1, subprogram 6, subprogram body 6.3, task 9, task body 9.1, task unit 9.1, terminate alternative 9.7.1, terminated task 9.4

5.2 Assignment Statement

1 An assignment statement replaces the current value of a variable with a new value specified by an expression. The named variable and the right-hand side expression must be of the same type; this type must not be a limited type.

2
```
assignment_statement ::=
    variable_name := expression;
```

3 For the execution of an assignment statement, the variable name and the expression are first evaluated, in some order that is not defined by the language. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is an array (the assignment then involves a subtype conversion as described in section 5.2.1). Finally, the value of the expression becomes the new value of the variable.[1]

4 The exception CONSTRAINT_ERROR is raised if the above-mentioned subtype check fails; in such a case the current value of the variable is left unchanged. If the variable is a subcomponent that depends on discriminants of an unconstrained record variable, then the execution of the assignment is erroneous if the value of any of these discriminants is changed by this execution.

5 **Examples:**
```
VALUE := MAX_VALUE - 1;
SHADE := BLUE;

NEXT_FRAME(F)(M, N) := 2.5;      -- see 4.1.1
U := DOT_PRODUCT(V, W);          -- see 6.5

WRITER := (STATUS     => OPEN,
           UNIT       => PRINTER,
           LINE_COUNT => 60);    -- see 3.7.3

NEXT_CAR.ALL := (72074, null);   -- see 3.8.1
```

---

[1] See also Appendix G, AI-00407.

6   **Examples of constraint checks:**

```
I, J : INTEGER range 1 .. 10;
K    : INTEGER range 1 .. 20;
 ...

I := J;  --  identical ranges
K := J;  --  compatible ranges
J := K;  --  will raise the exception CONSTRAINT_ERROR if K > 10
```

Notes:

7   The values of the discriminants of an object designated by an access value cannot be changed (not even by assigning a complete value to the object itself) since such objects, created by allocators, are always constrained (see 4.8); however, subcomponents of such objects may be unconstrained.

8   If the right-hand side expression is either a numeric literal or named number, or an attribute that yields a result of type *universal_integer* or *universal_real*, then an implicit type conversion is performed, as described in section 4.6.

9   The determination of the type of the variable of an assignment statement may require consideration of the expression if the variable name can be interpreted as the name of a variable designated by the access value returned by a function call, and similarly, as a component or slice of such a variable (see section 8.7 for the context of overload resolution).

10   **References:** access type 3.8, allocator 4.8, array 3.6, array assignment 5.2.1, component 3.6 3.7, constraint_error exception 11.1, designate 3.8, discriminant 3.7.1, erroneous 1.6, evaluation 4.5, expression 4.4, function call 6.4, implicit type conversion 4.6, name 4.1, numeric literal 2.4, object 3.2, overloading 6.6 8.7, slice 4.1.2, subcomponent 3.3, subtype 3.3, subtype conversion 4.6, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6, variable 3.2.1

5.2.1 Array Assignments

1   If the variable of an assignment statement is an array variable (including a slice variable), the value of the expression is implicitly converted to the subtype of the array variable; the result of this subtype conversion becomes the new value of the array variable.

2   This means that the new value of each component of the array variable is specified by the matching component in the array value obtained by evaluation of the expression (see 4.5.2 for the definition of matching components). The subtype conversion checks that for each component of the array variable there is a matching component in the array value, and vice versa. The exception CONSTRAINT_ERROR is raised if this check fails; in such a case the value of each component of the array variable is left unchanged.

3   **Examples:**
```
A : STRING(1 .. 31);
B : STRING(3 .. 33);
 ...

A := B;  --  same number of components

A(1 .. 9)  := "tar sauce";
A(4 .. 12) := A(1 .. 9);  --  A(1 .. 12) = "tartar sauce"
```
Notes:

4   Array assignment is defined even in the case of overlapping slices, because
the expression on the right-hand side is evaluated before performing any
component assignment. In the above example, an implementation yielding
A(1 .. 12) = "tartartartar" would be incorrect.

5   The implicit subtype conversion described above for assignment to an array
variable is performed only for the value of the right-hand side expression as a
whole; it is not performed for subcomponents that are array values.

6   **References:** array 3.6, assignment 5.2, constraint_error exception 11.1, matching
array components 4.5.2, slice 4.1.2, subtype conversion 4.6, type 3.3, variable 3.2.1

5.3 If Statements

1   An if statement selects for execution one or none of the enclosed sequences
of statements, depending on the (truth) value of one or more corresponding
conditions.

2
```
if_statement ::=
    if condition then
      sequence_of_statements
    {elsif condition then
      sequence_of_statements}
    [else
      sequence_of_statements]
     end if;
```
```
condition ::= boolean_expression
```

3   An expression specifying a condition must be of a boolean type.

4   For the execution of an if statement, the condition specified after **if**, and any
conditions specified after **elsif**, are evaluated in succession (treating a final
else as **elsif** TRUE **then**), until one evaluates to TRUE or all conditions
are evaluated and yield FALSE. If one condition evaluates to TRUE, then
the corresponding sequence of statements is executed; otherwise none of the
sequences of statements is executed.

Examples:

```
if MONTH = DECEMBER and DAY = 31 then
    MONTH := JANUARY;
    DAY   := 1;
    YEAR  := YEAR + 1;
end if;

if LINE_TOO_SHORT then
    raise LAYOUT_ERROR;
elsif LINE_FULL then
    NEW_LINE;
    PUT(ITEM);
else
    PUT(ITEM);
end if;

if MY_CAR.OWNER.VEHICLE /= MY_CAR then          -- see 3.8
    REPORT ("Incorrect data");
end if;
```

6 **References:** boolean type 3.5.3, evaluation 4.5, expression 4.4, sequence of statements 5.1

5.4 Case Statements

1 A case statement selects for execution one of a number of alternative sequences of statements; the chosen alternative is defined by the value of an expression.

```
2       case_statement ::=
          case expression is
               case_statement_alternative
             {case_statement_alternative}
          end case;

        case_statement_alternative ::=
          when choice {| choice } =>
               sequence_of_statements
```

3 The expression must be of a discrete type which must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type. Moreover, the type of this expression must not be a generic formal type. Each choice in a case statement alternative must be of the same type as the expression; the list of choices specifies for which values of the expression the alternative is chosen.[2]

---

[2] See also Appendix G, AI-00151.

4   If the expression is the name of an object whose subtype is static, then each value of this subtype must be represented once and only once in the set of choices of the case statement, and no other value is allowed; this rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a static subtype. Otherwise, for other forms of expression, each value of the (base) type of the expression must be represented once and only once in the set of choices, and no other value is allowed.

5   The simple expressions and discrete ranges given as choices in a case statement must be static. A choice defined by a discrete range stands for all values in the corresponding range (none if a null range). The choice **others** is only allowed for the last alternative and as its only choice; it stands for all values (possibly none) not given in the choices of previous alternatives. A component simple name is not allowed as a choice of a case statement alternative.

6   The execution of a case statement consists of the evaluation of the expression followed by the execution of the chosen sequence of statements.[3]

7   **Examples:**

```
case SENSOR is
   when ELEVATION  => RECORD_ELEVATION(SENSOR_VALUE);
   when AZIMUTH    => RECORD_AZIMUTH  (SENSOR_VALUE);
   when DISTANCE   => RECORD_DISTANCE (SENSOR_VALUE);
   when others     => null;
end case;

case TODAY is
   when MON        => COMPUTE_INITIAL_BALANCE;
   when FRI        => COMPUTE_CLOSING_BALANCE;
   when TUE .. THU => GENERATE_REPORT(TODAY);
   when SAT .. SUN => null;
end case;

case BIN_NUMBER(COUNT) is
   when 1      => UPDATE_BIN(1);
   when 2      => UPDATE_BIN(2);
   when 3 | 4  =>
      EMPTY_BIN(1);
      EMPTY_BIN(2);
   when others => raise ERROR;
end case;
```

---

[3] See also Appendix G, AI-00267.

Notes:

8 The execution of a case statement chooses one and only one alternative, since the choices are exhaustive and mutually exclusive. Qualification of the expression of a case statement by a static subtype can often be used to limit the number of choices that need be given explicitly.

9 An **others** choice is required in a case statement if the type of the expression is the type *universal_integer* (for example, if the expression is an integer literal), since this is the only way to cover all values of the type *universal_integer.*

10 **References:** base type 3.3, choice 3.7.3, context of overload resolution 8.7, discrete type 3.5, expression 4.4, function call 6.4, generic formal type 12.1, conversion 4.6, discrete type 3.5, enumeration literal 3.5.1, expression 4.4, name 4.1, object 3.2.1, overloading 6.6 8.7, qualified expression 4.7, sequence of statements 5.1, static discrete range 4.9, static subtype 4.9, subtype 3.3, type 3.3, type conversion 4.6, type mark 3.3.2

5.5 Loop Statements

1 A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times.

```
2    loop_statement ::=
        [loop_simple_name:]
           [iteration_scheme] loop
               sequence_of_statements
            end loop [loop_simple_name];

    iteration_scheme ::= while condition
       | for loop_parameter_specification

    loop_parameter_specification ::=
        identifier in [reverse] discrete_range
```

3 If a loop statement has a loop simple name, this simple name must be given both at the beginning and at the end.

4 A loop statement without an iteration scheme specifies repeated execution of the sequence of statements. Execution of the loop statement is complete when the loop is left as a consequence of the execution of an exit statement, or as a consequence of some other transfer of control (see 5.1).

5 For a loop statement with a **while** iteration scheme, the condition is evaluated before each execution of the sequence of statements; if the value of the condition is TRUE, the sequence of statements is executed, if FALSE the execution of the loop statement is complete.

6   For a loop statement with a **for** iteration scheme, the loop parameter specification is the declaration of the *loop parameter* with the given identifier. The loop parameter is an object whose type is the base type of the discrete range (see 3.6.1). Within the sequence of statements, the loop parameter is a constant. Hence a loop parameter is not allowed as the (left-hand side) variable of an assignment statement. Similarly the loop parameter must not be given as an **out** or **in out** parameter of a procedure or entry call statement, or as an **in out** parameter of a generic instantiation.[4]

7   For the execution of a loop statement with a **for** iteration scheme, the loop parameter specification is first elaborated. This elaboration creates the loop parameter and evaluates the discrete range.

8   If the discrete range is a null range, the execution of the loop statement is complete. Otherwise, the sequence of statements is executed once for each value of the discrete range (subject to the loop not being left as a consequence of the execution of an exit statement or as a consequence of some other transfer of control). Prior to each such iteration, the corresponding value of the discrete range is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

9   **Example of a loop statement without an iteration scheme:**
```
loop
   GET(CURRENT_CHARACTER);
   exit when CURRENT_CHARACTER = '*';
end loop;
```

10   **Example of a loop statement with a while iteration scheme:**
```
while BID(N).PRICE < CUT_OFF.PRICE loop
   RECORD_BID(BID(N).PRICE);
   N := N + 1;
end loop;
```

11   **Example of a loop statement with a for iteration scheme:**
```
for J in BUFFER'RANGE loop      -- legal even with a null range
   if BUFFER(J) /= SPACE then
      PUT(BUFFER(J));
   end if;
end loop;
```

---

[4] See also Appendix G, AI-00006.

12 **Example of a loop statement with a loop simple name:**

```
SUMMATION:
    while NEXT /= HEAD loop        -- see 3.8
        SUM  := SUM + NEXT.VALUE;
        NEXT := NEXT.SUCC;
    end loop SUMMATION;
```

Notes:

13 The scope of a loop parameter extends from the loop parameter specification to the end of the loop statement, and the visibility rules are such that a loop parameter is only visible within the sequence of statements of the loop.

14 The discrete range of a for loop is evaluated just once. Use of the reserved word **reverse** does not alter the discrete range, so that the following iteration schemes are not equivalent; the first has a null range.

```
for J in reverse 1 .. 0
for J in 0 .. 1
```

15 Loop names are also used in exit statements, and in expanded names (in a prefix of the loop parameter).

16 **References:** actual parameter 6.4.1, assignment statement 5.2, base type 3.3, bound of a range 3.5, condition 5.3, constant 3.2.1, context of overload resolution 8.7, conversion 4.6, declaration 3.1, discrete range 3.6.1, elaboration 3.1, entry call statement 9.5, evaluation 4.5, exit statement 5.7, expanded name 4.1.3, false boolean value 3.5.3, generic actual parameter 12.3, generic instantiation 12.3, goto statement 5.9, identifier 2.3, integer type 3.5.4, null range 3.5, object 3.2.1, prefix 4.1, procedure call 6.4, raising of exceptions 11, reserved word 2.9, return statement 5.8, scope 8.2, sequence of statements 5.1, simple name 4.1, terminate alternative 9.7.1, true boolean value 3.5.3 3.5.4, visibility 8.3

5.6 Block Statements

1 A block statement encloses a sequence of statements optionally preceded by a declarative part and optionally followed by exception handlers.

```
2    block_statement ::=
         [block_simple_name:]
            [declare
                 declarative_part]
             begin
                 sequence_of_statements
            [exception
                 exception_handler
                {exception_handler}]
            end [block_simple_name];
```

3 If a block statement has a block simple name, this simple name must be given both at the beginning and at the end.

4 The execution of a block statement consists of the elaboration of its declarative part (if any) followed by the execution of the sequence of statements. If the block statement has exception handlers, these service corresponding exceptions that are raised during the execution of the sequence of statements (see 11.2).

5 **Example:**
```
SWAP:
   declare
      TEMP : INTEGER;
   begin
      TEMP := V; V := U; U := TEMP;
   end SWAP;
```

Notes:

6 If task objects are declared within a block statement whose execution is completed, the block statement is not left until all its dependent tasks are terminated (see 9.4). This rule applies also to a completion caused by an exit, return, or goto statement; or by the raising of an exception.

7 Within a block statement, the block name can be used in expanded names denoting local entities such as SWAP.TEMP in the above example (see 4.1.3(f)).

8 **References:** declarative part 3.9, dependent task 9.4, exception handler 11.2, exit statement 5.7, expanded name 4.1.3, goto statement 5.9, raising of exceptions 11, return statement 5.8, sequence of statements 5.1, simple name 4.1, task object 9.2

5.7 Exit Statements

1 An exit statement is used to complete the execution of an enclosing loop statement (called the loop in what follows); the completion is conditional if the exit statement includes a condition.

2 ```
exit_statement ::=
    exit [loop_name] [when condition];
```
[5]

3  An exit statement with a loop name is only allowed within the named loop, and applies to that loop; an exit statement without a loop name is only allowed within a loop, and applies to the innermost enclosing loop (whether named or not). Furthermore, an exit statement that applies to a given loop must not appear within a subprogram body, package body, task body, generic body, or accept statement, if this construct is itself enclosed by the given loop.

_____

[5] See also Appendix G, AI-00210.

4    For the execution of an exit statement, the condition, if present, is first evaluated. Exit from the loop then takes place if the value is TRUE or if there is no condition.

5    **Examples:**

```
for N in 1 .. MAX_NUM_ITEMS loop
   GET_NEW_ITEM(NEW_ITEM);
   MERGE_ITEM(NEW_ITEM, STORAGE_FILE);
   exit when NEW_ITEM = TERMINAL_ITEM;
end loop;

MAIN_CYCLE:
   loop
      --  initial statements
      exit MAIN_CYCLE when FOUND;
      --  final statements
   end loop MAIN_CYCLE;
```

     Note:

6   Several nested loops can be exited by an exit statement that names the outer loop.

7   **References:** accept statement 9.5, condition 5.3, evaluation 4.5, generic body 12.1, loop name 5.5, loop statement 5.5, package body 7.1, subprogram body 6.3, true boolean value 3.5.3

5.8 Return Statements

1   A return statement is used to complete the execution of the innermost enclosing function, procedure, or accept statement.

2   `return_statement ::= return [expression];`

3   A return statement is only allowed within the body of a subprogram or generic subprogram, or within an accept statement, and applies to the innermost (enclosing) such construct; a return statement is not allowed within the body of a task unit, package, or generic package enclosed by this construct (on the other hand, it is allowed within a compound statement enclosed by this construct and, in particular, in a block statement).

4   A return statement for an accept statement or for the body of a procedure or generic procedure must not include an expression. A return statement for the body of a function or generic function must include an expression.

5   The value of the expression defines the result returned by the function. The type of this expression must be the base type of the type mark given after the reserved word **return** in the specification of the function or generic function (this type mark defines the result subtype).

6   For the execution of a return statement, the expression (if any) is first
      evaluated and a check is made that the value belongs to the result subtype.
      The execution of the return statement is thereby completed if the check
      succeeds; so also is the execution of the subprogram or of the accept statement.
      The exception CONSTRAINT_ERROR is raised at the place of the return
      statement if the check fails.

7   **Examples:**

```
return;                        -- in a procedure
return KEY_VALUE(LAST_INDEX);  -- in a function
```

      Note:

8   If the expression is either a numeric literal or named number, or an attribute
      that yields a result of type *universal_integer* or *universal_real*, then an implicit
      conversion of the result is performed as described in section 4.6.

9   **References:** accept statement 9.5, attribute A, block statement 5.6, constraint_error
      exception 11.1, expression 4.4, function body 6.3, function call 6.4, generic body 12.1,
      implicit type conversion 4.6, named number 3.2, numeric literal 2.4, package body
      7.1, procedure body 6.3, reserved word 2.9, result subtype 6.1, subprogram body 6.3,
      subprogram specification 6.1, subtype 3.3, task body 9.1, type mark 3.3.2, universal_
      integer type 3.5.4, universal_real type 3.5.6

5.9 Goto Statements

1   A goto statement specifies an explicit transfer of control from this statement to
      a *target* statement named by a label.

2   goto_statement ::= **goto** *label*_name;

3   The innermost sequence of statements that encloses the target statement
      must also enclose the goto statement (note that the goto statement can
      be a statement of an inner sequence). Furthermore, if a goto statement is
      enclosed by an accept statement or the body of a program unit, then the target
      statement must not be outside this enclosing construct; conversely, it follows
      from the previous rule that if the target statement is enclosed by such a
      construct, then the goto statement cannot be outside.

4   The execution of a goto statement transfers control to the named target
      statement.

      Note:

5   The above rules allow transfer of control to a statement of an enclosing
      sequence of statements but not the reverse. Similarly, they prohibit transfers
      of control such as between alternatives of a case statement, if statement, or
      select statement; between exception handlers; or from an exception handler of
      a frame back to the sequence of statements of this frame.

₆ **Example:**

```
<<COMPARE>>
    if A(I) < ELEMENT then
        if LEFT(I) /= 0 then
            I := LEFT(I);
            goto COMPARE;
        end if;
        --  some statements
    end if;
```

₇ **References:** accept statement 9.5, block statement 5.6, case statement 5.4, compound statement 5.1, exception handler 11.2, frame 11.2, generic body 12.1, if statement 5.3, label 5.1, package body 7.1, program unit 6, select statement 9.7, sequence of statements 5.1, statement 5.1, subprogram body 6.3, task body 9.1, transfer of control 5.1

6

Subprograms

1. Subprograms are one of the four forms of *program unit*, of which programs can be composed. The other forms are packages, task units, and generic units.

2. A subprogram is a program unit whose execution is invoked by a subprogram call. There are two forms of subprogram: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution.

3. **References:** function 6.5, function call 6.4, generic unit 12, package 7, procedure 6.1, procedure call 6.4, subprogram body 6.3, subprogram call 6.4, subprogram declaration 6.1, task unit 9

6.1 Subprogram Declarations

1. A subprogram declaration declares a procedure or a function, as indicated by the initial reserved word.

2.
```
subprogram_declaration ::= subprogram_specification;

subprogram_specification ::=
    procedure identifier [formal_part]
  | function designator  [formal_part] return type_mark

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

formal_part ::=
   (parameter_specification {; parameter_specification})

parameter_specification ::=
   identifier_list : mode type_mark [:= expression]

mode ::= [in] | in out | out
```

3   The specification of a procedure specifies its identifier and its *formal parameters* (if any). The specification of a function specifies its designator, its formal parameters (if any) and the subtype of the returned value (the *result subtype*). A designator that is an operator symbol is used for the overloading of an operator. The sequence of characters represented by an operator symbol must be an operator belonging to one of the six classes of overloadable operators defined in section 4.5 (extra spaces are not allowed and the case of letters is not significant).

4   A parameter specification with several identifiers is equivalent to a sequence of single parameter specifications, as explained in section 3.2. Each single parameter specification declares a formal parameter. If no mode is explicitly given, the mode **in** is assumed. If a parameter specification ends with an expression, the expression is the *default expression* of the formal parameter. A default expression is only allowed in a parameter specification if the mode is **in** (whether this mode is indicated explicitly or implicitly). The type of a default expression must be that of the corresponding formal parameter.

5   The use of a name that denotes a formal parameter is not allowed in default expressions of a formal part if the specification of the parameter is itself given in this formal part.

6   The elaboration of a subprogram declaration elaborates the corresponding formal part. The elaboration of a formal part has no other effect.

7   **Examples of subprogram declarations:**
```
procedure TRAVERSE_TREE;
procedure INCREMENT(X : in out INTEGER);
procedure RIGHT_INDENT(MARGIN : out LINE_SIZE); --  see 3.5.4
procedure SWITCH(FROM, TO : in out LINK);        -- see 3.8.1

function RANDOM return PROBABILITY;                -- see 3.5.7

function MIN_CELL(X : LINK) return CELL;      -- see 3.8.1
function NEXT_FRAME(K : POSITIVE) return FRAME; --  see 3.8
function DOT_PRODUCT(LEFT,RIGHT: VECTOR) return REAL;  -- see 3.6

function "*"(LEFT,RIGHT : MATRIX) return MATRIX;      -- see 3.6
```

8   **Examples of in parameters with default expressions:**
```
procedure PRINT_HEADER(PAGES  : in NATURAL;
                       HEADER : in LINE
                           := (1 .. LINE'LAST => ' '); --  see 3.6
                       CENTER : in BOOLEAN
                           := TRUE);
```

Notes:

9 The evaluation of default expressions is caused by certain subprogram calls, as described in section 6.4.2 (default expressions are not evaluated during the elaboration of the subprogram declaration).

10 All subprograms can be called recursively and are reentrant.

11 **References:** declaration 3.1, elaboration 3.9, evaluation 4.5, expression 4.4, formal parameter 6.2, function 6.5, identifier 2.3, identifier list 3.2, mode 6.2, name 4.1, elaboration has no other effect 3.9, operator 4.5, overloading 6.6 8.7, procedure 6, string literal 2.6, subprogram call 6.4, type mark 3.3.2

6.2 Formal Parameter Modes

1 The value of an object is said to be *read* when this value is evaluated; it is also said to be read when one of its subcomponents is read. The value of a variable is said to be *updated* when an assignment is performed to the variable, and also (indirectly) when the variable is used as actual parameter of a subprogram call or entry call statement that updates its value; it is also said to be updated when one of its subcomponents is updated.

2 A formal parameter of a subprogram has one of the three following modes:

3
in The formal parameter is a constant and permits only reading of the value of the associated actual parameter.

4
in out The formal parameter is a variable and permits both reading and updating of the value of the associated actual parameter.

5
out The formal parameter is a variable and permits updating of the value of the associated actual parameter.

The value of a scalar parameter that is not updated by the call is undefined upon return; the same holds for the value of a scalar subcomponent, other than a discriminant. Reading the bounds and discriminants of the formal parameter and of its subcomponents is allowed, but no other reading.

6 For a scalar parameter, the above effects are achieved by copy: at the start of each call, if the mode is **in** or **in out**, the value of the actual parameter is copied into the associated formal parameter; then after normal completion of the subprogram body, if the mode is **in out** or **out**, the value of the formal parameter is copied back into the associated actual parameter. For a parameter whose type is an access type, copy-in is used for all three modes, and copy-back for the modes **in out** and **out.**

7   For a parameter whose type is an array, record, or task type, an implementation may likewise achieve the above effects by copy, as for scalar types. In addition, if copy is used for a parameter of mode **out**, then copy-in is required at least for the bounds and discriminants of the actual parameter and of its subcomponents, and also for each subcomponent whose type is an access type. Alternatively, an implementation may achieve these effects by reference, that is, by arranging that every use of the formal parameter (to read or to update its value) be treated as a use of the associated actual parameter, throughout the execution of the subprogram call. The language does not define which of these two mechanisms is to be adopted for parameter passing, nor whether different calls to the same subprogram are to use the same mechanism. The execution of a program is erroneous if its effect depends on which mechanism is selected by the implementation.

The DEC Ada run-time reference manuals describe the parameter passing mechanisms used by DEC Ada on OpenVMS and Digital UNIX systems.

8   For a parameter whose type is a private type, the above effects are achieved according to the rule that applies to the corresponding full type declaration.

9   Within the body of a subprogram, a formal parameter is subject to any constraint resulting from the type mark given in its parameter specification. For a formal parameter of an unconstrained array type, the bounds are obtained from the actual parameter, and the formal parameter is constrained by these bounds (see 3.6.1). For a formal parameter whose declaration specifies an unconstrained (private or record) type with discriminants, the discriminants of the formal parameter are initialized with the values of the corresponding discriminants of the actual parameter; the formal parameter is unconstrained if and only if the mode is **in out** or **out** and the variable name given for the actual parameter denotes an unconstrained variable (see 3.7.1 and 6.4.1).

10   If the actual parameter of a subprogram call is a subcomponent that depends on discriminants of an unconstrained record variable, then the execution of the call is erroneous if the value of any of the discriminants of the variable is changed by this execution; this rule does not apply if the mode is **in** and the type of the subcomponent is a scalar type or an access type.

Notes:

11   For parameters of array and record types, the parameter passing rules have these consequences:

12   • If the execution of a subprogram is abandoned as a result of an exception, the final value of an actual parameter of such a type can be either its value before the call or a value assigned to the formal parameter during the execution of the subprogram.

13   • If no actual parameter of such a type is accessible by more than one path, then the effect of a subprogram call (unless abandoned) is the same whether or not the implementation uses copying for parameter passing. If, however, there are multiple access paths to such a parameter (for example, if a global variable, or another formal parameter, refers to the same actual parameter), then the value of the formal is undefined after updating the actual other than by updating the formal. A program using such an undefined value is erroneous.

14   The same parameter modes are defined for formal parameters of entries (see 9.5) with the same meaning as for subprograms. Different parameter modes are defined for generic formal parameters (see 12.1.1).

15   For all modes, if an actual parameter designates a task, the associated formal parameter designates the same task; the same holds for a subcomponent of an actual parameter and the corresponding subcomponent of the associated formal parameter.

16   **References:** access type 3.8, actual parameter 6.4.1, array type 3.6, assignment 5.2, bound of an array 3.6.1, constraint 3.3, depend on a discriminant 3.7.1, discriminant 3.7.1, entry call statement 9.5, erroneous 1.6, evaluation 4.5, exception 11, expression 4.4, formal parameter 6.1, generic formal parameter 12.1, global 8.1, mode 6.1, null access value 3.8, object 3.2, parameter specification 6.1, private type 7.4, record type 3.7, scalar type 3.5, subcomponent 3.3, subprogram body 6.3, subprogram call statement 6.4, task 9, task type 9.2, type mark 3.3.2, unconstrained array type 3.6, unconstrained type with discriminants 3.7.1, unconstrained variable 3.2.1, variable 3.2.1

6.3 Subprogram Bodies

1   A subprogram body specifies the execution of a subprogram.

2
```
subprogram_body ::=
    subprogram_specification is
        [declarative_part]
    begin
        sequence_of_statements
    [exception
        exception_handler
        {exception_handler}]
    end [designator];
```

3   The declaration of a subprogram is optional. In the absence of such a declaration, the subprogram specification of the subprogram body (or body stub) acts as the declaration. For each subprogram declaration, there must be a corresponding body (except for a subprogram written in another language, as explained in section 13.9). If both a declaration and a body are given,

the subprogram specification of the body must conform to the subprogram specification of the declaration (see section 6.3.1 for conformance rules).

4   If a designator appears at the end of a subprogram body, it must repeat the designator of the subprogram specification.

5   The elaboration of a subprogram body has no other effect than to establish that the body can from then on be used for the execution of calls of the subprogram.

6   The execution of a subprogram body is invoked by a subprogram call (see 6.4). For this execution, after establishing the association between formal parameters and actual parameters, the declarative part of the body is elaborated, and the sequence of statements of the body is then executed. Upon completion of the body, return is made to the caller (and any necessary copying back of formal to actual parameters occurs (see 6.2)). The optional exception handlers at the end of a subprogram body handle exceptions raised during the execution of the sequence of statements of the subprogram body (see 11.4).

Note:

7   It follows from the visibility rules that if a subprogram declared in a package is to be visible outside the package, a subprogram specification must be given in the visible part of the package. The same rules dictate that a subprogram declaration must be given if a call of the subprogram occurs textually before the subprogram body (the declaration must then occur earlier than the call in the program text). The rules given in sections 3.9 and 7.1 imply that a subprogram declaration and the corresponding body must both occur immediately within the same declarative region.

8   **Example of subprogram body:**

```
procedure PUSH(E : in ELEMENT_TYPE; S : in out STACK) is
begin
    if S.INDEX = S.SIZE then
        raise STACK_OVERFLOW;
    else
        S.INDEX := S.INDEX + 1;
        S.SPACE(S.INDEX) := E;
    end if;
end PUSH;
```

9   **References:** actual parameter 6.4.1, body stub 10.2, conform 6.3.1, declaration 3.1, declarative part 3.9, declarative region 8.1, designator 6.1, elaboration 3.9, elaboration has no other effect 3.1, exception 11, exception handler 11.2, formal parameter 6.1, occur immediately within 8.1, package 7, sequence of statements 5.1, subprogram 6, subprogram call 6.4, subprogram declaration 6.1, subprogram specification 6.1, visibility 8.3, visible part 7.2

6.3.1 Conformance Rules

1 Whenever the language rules require or allow the specification of a given subprogram to be provided in more than one place, the following variations are allowed at each place:

2 • A numeric literal can be replaced by a different numeric literal if and only if both have the same value.

3 • A simple name can be replaced by an expanded name in which this simple name is the selector, if and only if at both places the meaning of the simple name is given by the same declaration.

4 • A string literal given as an operator symbol can be replaced by a different string literal if and only if both represent the same operator.[1]

5 Two subprogram specifications are said to *conform* if, apart from comments and the above allowed variations, both specifications are formed by the same sequence of lexical elements, and corresponding lexical elements are given the same meaning by the visibility and overloading rules.[2]

6 Conformance is likewise defined for formal parts, discriminant parts, and type marks (for deferred constants and for actual parameters that have the form of a type conversion (see 6.4.1)).

Notes:

7 A simple name can be replaced by an expanded name even if the simple name is itself the prefix of a selected component. For example, Q.R can be replaced by P.Q.R if Q is declared immediately within P.

8 The following specifications do not conform since they are not formed by the same sequence of lexical elements:

```
procedure P(X,Y : INTEGER)
procedure P(X : INTEGER; Y : INTEGER)
procedure P(X,Y : in INTEGER)
```

9 **References:** actual parameter 6.4 6.4.1, allow 1.6, comment 2.7, declaration 3.1, deferred constant 7.4.3, direct visibility 8.3, discriminant part 3.7.1, expanded name 4.1.3, formal part 6.1, lexical element 2, name 4.1, numeric literal 2.4, operator symbol 6.1, overloading 6.6 8.7, prefix 4.1, selected component 4.1.3, selector 4.1.3, simple name 4.1, subprogram specification 6.1, type conversion 4.6, visibility 8.3

---

[1] See also Appendix G, AI-00493.
[2] See also Appendix G, AI-00350.

6.3.2 Inline Expansion of Subprograms

1   The pragma INLINE is used to indicate that inline expansion of the subprogram body is desired for every call of each of the named subprograms. The form of this pragma is as follows:

2   **pragma** INLINE (name {, name});

Each name is either the name of a subprogram or the name of a generic subprogram. The pragma INLINE is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

3   If the pragma appears at the place of a declarative item, each name must denote a subprogram or a generic subprogram declared by an earlier declarative item of the same declarative part or package specification. If several (overloaded) subprograms satisfy this requirement, the pragma applies to all of them. If the pragma appears after a given library unit, the only name allowed is the name of this unit. If the name of a generic subprogram is mentioned in the pragma, this indicates that inline expansion is desired for calls of all subprograms obtained by instantiation of the named generic unit.[3]

In DEC Ada, the subprogram name must be an identifier or a string literal that denotes an operator symbol.

Also, if the name specified by a pragma INLINE is declared by a renaming declaration, the pragma INLINE applies to the subprogram only if the declaration of the subprogram that has been renamed, the renaming declaration, and the pragma all occur in the same declarative part or package specification. The pragma is ignored if these conditions are not satisfied.

4   The meaning of a subprogram is not changed by the pragma INLINE. For each call of the named subprograms, an implementation is free to follow or to ignore the recommendation expressed by the pragma. (Note, in particular, that the recommendation cannot generally be followed for a recursive subprogram.)

In DEC Ada, a call of a subprogram for which the pragma INLINE has been specified is expanded inline provided that certain conditions are satisfied. These conditions are given in the DEC Ada run-time reference manuals. The same criteria apply to subprograms that result from instantiation of a generic declaration for which a pragma INLINE was specified.

---

[3]   See also Appendix G, AI-00200 and AI-00242.

Notes:

The meaning of the subprogram name is determined as for any name (see 8.3), except that the name can denote more than one subprogram. In the following declaration, the pragma INLINE applies to the first two procedures. It does not apply to the third because the declaration is not visible at the place of the pragma:

```
procedure P (B: BOOLEAN);
procedure P (I: INTEGER);
pragma INLINE (P);
procedure P (F: FLOAT);
```

If a pragma INLINE and a pragma INTERFACE are used together, the pragma INLINE is ignored regardless of the order in which the two pragmas appear.

Example of renaming:
```
package CHOOSE_R is
    procedure P (X: INTEGER);
    procedure P (X: FLOAT);
private
    procedure R (X: FLOAT) renames P;
    pragma INLINE(R); -- second procedure P will be expanded
                      -- inline when procedure R is called
end CHOOSE_R;
```

5   **References:** allow 1.6, compilation 10.1, compilation unit 10.1, declarative item 3.9, declarative part 3.9, generic subprogram 12.1, generic unit 12 12.1, instantiation 12.3, library unit 10.1, name 4.1, overloading 6.6 8.7, package specification 7.1, pragma 2.8, subprogram 6, subprogram body 6.3, subprogram call 6.4

identifier 2.3, operator symbol 6.1, renaming declaration 8.5, string literal 2.6

6.4 Subprogram Calls

1   A subprogram call is either a procedure call statement or a function call; it invokes the execution of the corresponding subprogram body. The call specifies the association of the actual parameters, if any, with formal parameters of the subprogram.

2
```
procedure_call_statement ::=
    procedure_name [actual_parameter_part];

function_call ::=
    function_name [actual_parameter_part]

actual_parameter_part ::=
    (parameter_association {, parameter_association})

parameter_association ::=
    [formal_parameter =>] actual_parameter
```

```
formal_parameter ::= parameter_simple_name

actual_parameter ::=
    expression | variable_name | type_mark(variable_name)
```

3   Each parameter association associates an actual parameter with a
     corresponding formal parameter. A parameter association is said to be
     named if the formal parameter is named explicitly; it is otherwise said to be
     positional. For a positional association, the actual parameter corresponds to
     the formal parameter with the same position in the formal part.

4   Named associations can be given in any order, but if both positional and named
     associations are used in the same call, positional associations must occur first,
     at their normal position. Hence once a named association is used, the rest of
     the call must use only named associations.

5   For each formal parameter of a subprogram, a subprogram call must specify
     exactly one corresponding actual parameter. This actual parameter is specified
     either explicitly, by a parameter association, or, in the absence of such an
     association, by a default expression (see 6.4.2).

6   The parameter associations of a subprogram call are evaluated in some order
     that is not defined by the language. Similarly, the language rules do not define
     in which order the values of **in out** or **out** parameters are copied back into the
     corresponding actual parameters (when this is done).

7   **Examples of procedure calls:**
```
TRAVERSE_TREE;                    -- see 6.1
TABLE_MANAGER.INSERT(E);          -- see 7.5
PRINT_HEADER(128, TITLE, TRUE);   -- see 6.1

SWITCH(FROM => X, TO => NEXT);    -- see 6.1

PRINT_HEADER(128,
            HEADER => TITLE,
            CENTER => TRUE);      -- see 6.1

PRINT_HEADER(HEADER => TITLE,
            CENTER => TRUE,
            PAGES  => 128);       -- see 6.1
```

8   **Examples of function calls:**
```
DOT_PRODUCT(U, V)    -- see 6.1 and 6.5
CLOCK                -- see 9.6
```

9   **References:** default expression for a formal parameter 6.1, erroneous 1.6, expression
     4.4, formal parameter 6.1, formal part 6.1, name 4.1, simple name 4.1, subprogram 6,
     type mark 3.3.2, variable 3.2.1

6.4.1 Parameter Associations

1 Each actual parameter must have the same type as the corresponding formal parameter.

2 An actual parameter associated with a formal parameter of mode **in** must be an expression; it is evaluated before the call.

3 An actual parameter associated with a formal parameter of mode **in out** or **out** must be either the name of a variable, or of the form of a type conversion whose argument is the name of a variable. In either case, for the mode **in out**, the variable must not be a formal parameter of mode **out** or a subcomponent thereof. For an actual parameter that has the form of a type conversion, the type mark must conform (see 6.3.1) to the type mark of the formal parameter; the allowed operand and target types are the same as for type conversions (see 4.6).[4]

4 The variable name given for an actual parameter of mode **in out** or **out** is evaluated before the call. If the actual parameter has the form of a type conversion, then before the call, for a parameter of mode **in out**, the variable is converted to the specified type; after (normal) completion of the subprogram body, for a parameter of mode **in out** or **out**, the formal parameter is converted back to the type of the variable. (The type specified in the conversion must be that of the formal parameter.)[5]

5 The following constraint checks are performed for parameters of scalar and access types:

6 • Before the call: for a parameter of mode **in** or **in out**, it is checked that the value of the actual parameter belongs to the subtype of the formal parameter.

7 • After (normal) completion of the subprogram body: for a parameter of mode **in out** or **out**, it is checked that the value of the formal parameter belongs to the subtype of the actual variable. In the case of a type conversion, the value of the formal parameter is converted back and the check applies to the result of the conversion.

8 In each of the above cases, the execution of the program is erroneous if the checked value is undefined.

9 For other types, for all modes, a check is made before the call as for scalar and access types; no check is made upon return.[6]

---

[4] See also Appendix G, AI-00245.
[5] See also Appendix G, AI-00024.
[6] See also Appendix G, AI-00025 and AI-00396.

10   The exception CONSTRAINT_ERROR is raised at the place of the subprogram call if either of these checks fails.

Note:

11   For array types and for types with discriminants, the check before the call is sufficient (a check upon return would be redundant) if the type mark of the formal parameter denotes a constrained subtype, since neither array bounds nor discriminants can then vary.

12   If this type mark denotes an unconstrained array type, the formal parameter is constrained with the bounds of the corresponding actual parameter and no check (neither before the call nor upon return) is needed (see 3.6.1). Similarly, no check is needed if the type mark denotes an unconstrained type with discriminants, since the formal parameter is then constrained exactly as the corresponding actual parameter (see 3.7.1).

13   **References:** actual parameter 6.4, array bound 3.6, array type 3.6, call of a subprogram 6.4, conform 6.3.1, constrained subtype 3.3, constraint 3.3, constraint_error exception 11.1, discriminant 3.7.1, erroneous 1.6, evaluation 4.5, evaluation of a name 4.1, expression 4.4, formal parameter 6.1, mode 6.1, name 4.1, parameter association 6.4, subtype 3.3, type 3.3, type conversion 4.6, type mark 3.3.2, unconstrained array type 3.6, unconstrained type with discriminants 3.7.1, undefined value 3.2.1, variable 3.2.1

6.4.2 Default Parameters

1   If a parameter specification includes a default expression for a parameter of mode **in**, then corresponding subprogram calls need not include a parameter association for the parameter. If a parameter association is thus omitted from a call, then the rest of the call, following any initial positional associations, must use only named associations.

2   For any omitted parameter association, the default expression is evaluated before the call and the resulting value is used as an implicit actual parameter.

3   **Examples of procedures with default values:**

```
procedure ACTIVATE(PROCESS : in PROCESS_NAME;
                   AFTER   : in PROCESS_NAME := NO_PROCESS;
                   WAIT    : in DURATION := 0.0;
                   PRIOR   : in BOOLEAN := FALSE);

procedure PAIR(LEFT, RIGHT : PERSON_NAME := new PERSON);
```

4   **Examples of their calls:**
```
ACTIVATE(X);
ACTIVATE(X, AFTER => Y);
ACTIVATE(X, WAIT => 60.0, PRIOR => TRUE);
ACTIVATE(X, Y, 10.0, FALSE);

PAIR;
PAIR(LEFT => new PERSON, RIGHT => new PERSON);
```

Note:

5   If a default expression is used for two or more parameters in a multiple
     parameter specification, the default expression is evaluated once for each
     omitted parameter. Hence in the above examples, the two calls of PAIR are
     equivalent.

6   **References:** actual parameter 6.4.1, default expression for a formal parameter 6.1,
     evaluation 4.5, formal parameter 6.1, mode 6.1, named parameter association 6.4,
     parameter association 6.4, parameter specification 6.1, positional parameter association
     6.4, subprogram call 6.4

6.5 Function Subprograms

1   A function is a subprogram that returns a value (the result of the function
     call). The specification of a function starts with the reserved word **function**,
     and the parameters, if any, must have the mode **in** (whether this mode
     is specified explicitly or implicitly). The statements of the function body
     (excluding statements of program units that are inner to the function body)
     must include one or more return statements specifying the returned value.

2   The exception PROGRAM_ERROR is raised if a function body is left otherwise
     than by a return statement. This does not apply if the execution of the function
     is abandoned as a result of an exception.

3   **Example:**
```
function DOT_PRODUCT(LEFT, RIGHT : VECTOR) return REAL is
   SUM : REAL := 0.0;
begin
   CHECK(LEFT'FIRST = RIGHT'FIRST and LEFT'LAST = RIGHT'LAST);
   for J in LEFT'RANGE loop
      SUM := SUM + LEFT(J)*RIGHT(J);
   end loop;
   return SUM;
end DOT_PRODUCT;
```

4   **References:** exception 11, formal parameter 6.1, function 6.1, function body 6.3,
     function call 6.4, function specification 6.1, mode 6.1, program_error exception 11.1,
     raising of exceptions 11, return statement 5.8, statement 5

6.6 Parameter and Result Type Profile—Overloading of Subprograms

1 Two formal parts are said to have the same *parameter type profile* if and only if they have the same number of parameters, and at each parameter position corresponding parameters have the same base type. A subprogram or entry has the same *parameter and result type profile* as another subprogram or entry if and only if both have the same parameter type profile, and either both are functions with the same result base type, or neither of the two is a function.

2 The same subprogram identifier or operator symbol can be used in several subprogram specifications. The identifier or operator symbol is then said to be *overloaded*; the subprograms that have this identifier or operator symbol are also said to be overloaded and to overload each other. As explained in section 8.3, if two subprograms overload each other, one of them can hide the other only if both subprograms have the same parameter and result type profile (see section 8.3 for the other requirements that must be met for hiding).

3 A call to an overloaded subprogram is ambiguous (and therefore illegal) if the name of the subprogram, the number of parameter associations, the types and the order of the actual parameters, the names of the formal parameters (if named associations are used), and the result type (for functions) are not sufficient to determine exactly one (overloaded) subprogram specification.

4 **Examples of overloaded subprograms:**

```
procedure PUT(X : INTEGER);
procedure PUT(X : STRING);

procedure SET(TINT   : COLOR);
procedure SET(SIGNAL : LIGHT);
```

5 **Examples of calls:**

```
PUT(28);
PUT("no possible ambiguity here");

SET(TINT   => RED);
SET(SIGNAL => RED);
SET(COLOR'(RED));

--  SET(RED) would be ambiguous since RED may
--  denote a value either of type COLOR or of type LIGHT
```

Notes:

6   The notion of parameter and result type profile does not include parameter names, parameter modes, parameter subtypes, default expressions and their presence or absence.

7   Ambiguities may (but need not) arise when actual parameters of the call of an overloaded subprogram are themselves overloaded function calls, literals, or aggregates. Ambiguities may also (but need not) arise when several overloaded subprograms belonging to different packages are visible. These ambiguities can usually be resolved in several ways: qualified expressions can be used for some or all actual parameters, and for the result, if any; the name of the subprogram can be expressed more explicitly as an expanded name; finally, the subprogram can be renamed.

8   **References:** actual parameter 6.4.1, aggregate 4.3, base type 3.3, default expression for a formal parameter 6.1, entry 9.5, formal parameter 6.1, function 6.5, function call 6.4, hiding 8.3, identifier 2.3, illegal 1.6, literal 4.2, mode 6.1, named parameter association 6.4, operator symbol 6.1, overloading 8.7, package 7, parameter of a subprogram 6.2, qualified expression 4.7, renaming declaration 8.5, result subtype 6.1, subprogram 6, subprogram specification 6.1, subtype 3.3, type 3.3

6.7 Overloading of Operators

1   The declaration of a function whose designator is an operator symbol is used to overload an operator. The sequence of characters of the operator symbol must be either a logical, a relational, a binary adding, a unary adding, a multiplying, or a highest precedence operator (see 4.5). Neither membership tests nor the short-circuit control forms are allowed as function designators.

2   The subprogram specification of a unary operator must have a single parameter. The subprogram specification of a binary operator must have two parameters; for each use of this operator, the first parameter takes the left operand as actual parameter, the second parameter takes the right operand. Similarly, a generic function instantiation whose designator is an operator symbol is only allowed if the specification of the generic function has the corresponding number of parameters. Default expressions are not allowed for the parameters of an operator (whether the operator is declared with an explicit subprogram specification or by a generic instantiation).

3   For each of the operators "+" and "−", overloading is allowed both as a unary and as a binary operator.

4   The explicit declaration of a function that overloads the equality operator "=", other than by a renaming declaration, is only allowed if both parameters are of the same limited type. An overloading of equality must deliver a result of the predefined type BOOLEAN; it also implicitly overloads the inequality operator "/=" so that this still gives the complementary result to the equality operator. Explicit overloading of the inequality operator is not allowed.

5   A renaming declaration whose designator is the equality operator is only allowed to rename another equality operator. (For example, such a renaming declaration can be used when equality is visible by selection but not directly visible.)

Note:

6   Overloading of relational operators does not affect basic comparisons such as testing for membership in a range or the choices in a case statement.

7   **Examples:**
```
function "+" (LEFT, RIGHT : MATRIX) return MATRIX;
function "+" (LEFT, RIGHT : VECTOR) return VECTOR;

--  assuming that A, B, and C are of the type VECTOR
--  the three following assignments are equivalent

A := B + C;

A := "+"(B, C);
A := "+"(LEFT => B, RIGHT => C);
```

8   **References:** allow 1.6, actual parameter 6.4.1, binary adding operator 4.5 4.5.3, boolean predefined type 3.5.3, character 2.1, complementary result 4.5.2, declaration 3.1, default expression for a formal parameter 6.1, designator 6.1, directly visible 8.3, equality operator 4.5, formal parameter 6.1, function declaration 6.1, highest precedence operator 4.5 4.5.6, implicit declaration 3.1, inequality operator 4.5.2, limited type 7.4.4, logical operator 4.5 4.5.1, membership test 4.5 4.5.2, multiplying operator 4.5 4.5.5, operator 4.5, operator symbol 6.1, overloading 6.6 8.7, relational operator 4.5 4.5.2, short-circuit control form 4.5 4.5.1, type definition 3.3.1, unary adding operator 4.5 4.5.4, visible by selection 8.3

7
Packages

1   Packages are one of the four forms of program unit, of which programs can be composed. The other forms are subprograms, task units, and generic units.

2   Packages allow the specification of groups of logically related entities. In their simplest form packages specify pools of common object and type declarations. More generally, packages can be used to specify groups of related entities including also subprograms that can be called from outside the package, while their inner workings remain concealed and protected from outside users.

3   **References:** generic unit 12, program unit 6, subprogram 6, task unit 9, type declaration 3.3.1

7.1 Package Structure

1   A package is generally provided in two parts: a package specification and a package body. Every package has a package specification, but not all packages have a package body.

2
```
package_declaration ::= package_specification;

package_specification ::=
    package identifier is
      {basic_declarative_item}
  [private
      {basic_declarative_item}]
   end [package_simple_name]

package_body ::=
    package body package_simple_name is
        [declarative_part]
  [begin
        sequence_of_statements
  [exception
        exception_handler
      {exception_handler}]]
   end [package_simple_name];
```

³ The simple name at the start of a package body must repeat the package identifier. Similarly if a simple name appears at the end of the package specification or body, it must repeat the package identifier.

⁴ If a subprogram declaration, a package declaration, a task declaration, or a generic declaration is a declarative item of a given package specification, then the body (if there is one) of the program unit declared by the declarative item must itself be a declarative item of the declarative part of the body of the given package.

Notes:

⁵ A simple form of package, specifying a pool of objects and types, does not require a package body. One of the possible uses of the sequence of statements of a package body is to initialize such objects. For each subprogram declaration there must be a corresponding body (except for a subprogram written in another language, as explained in section 13.9). If the body of a program unit is a body stub, then a separately compiled subunit containing the corresponding proper body is required for the program unit (see 10.2). A body is not a basic declarative item and so cannot appear in a package specification.

⁶ A package declaration is either a library package (see 10.2) or a declarative item declared within another program unit.

⁷ **References:** basic declarative item 3.9, body stub 10.2, declarative item 3.9, declarative part 3.9, exception handler 11.2, generic body 12.2, generic declaration 12.1, identifier 2.3, library unit 10.1, object 3.2, package body 7.3, program unit 6, proper body 3.9, sequence of statements 5.1, simple name 4.1, subprogram body 6.3, subprogram declaration 6.1, subunit 10.2, task body 9.1, task declaration 9.1, type 3.3

7.2 Package Specifications and Declarations

¹ The first list of declarative items of a package specification is called the *visible part* of the package. The optional list of declarative items after the reserved word **private** is called the *private part* of the package.

² An entity declared in the private part of a package is not visible outside the package itself (a name denoting such an entity is only possible within the package). In contrast, expanded names denoting entities declared in the visible part can be used even outside the package; furthermore, direct visibility of such entities can be achieved by means of use clauses (see 4.1.3 and 8.4).

³ The elaboration of a package declaration consists of the elaboration of its basic declarative items in the given order.

Notes:

4   The visible part of a package contains all the information that another program unit is able to know about the package. A package consisting of only a package specification (that is, without a package body) can be used to represent a group of common constants or variables, or a common pool of objects and types, as in the examples below.

5   **Example of a package describing a group of common variables:**
```
package PLOTTING_DATA is
    PEN_UP : BOOLEAN;

    CONVERSION_FACTOR,
    X_OFFSET, Y_OFFSET,
    X_MIN,    Y_MIN,
    X_MAX,    Y_MAX:   REAL;      -- see 3.5.7

    X_VALUE : array (1 .. 500) of REAL;
    Y_VALUE : array (1 .. 500) of REAL;
end PLOTTING_DATA;
```

6   **Example of a package describing a common pool of objects and types:**
```
package WORK_DATA is
    type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
    type HOURS_SPENT is delta 0.25 range 0.0 .. 24.0;
    type TIME_TABLE  is array (DAY) of HOURS_SPENT;

    WORK_HOURS   : TIME_TABLE;
    NORMAL_HOURS : constant TIME_TABLE := (MON .. THU => 8.25,
                                           FRI       => 7.0,
                                           SAT | SUN => 0.0);

end WORK_DATA;
```

7   **References:** basic declarative item 3.9, constant 3.2.1, declarative item 3.9, direct visibility 8.3, elaboration 3.9, expanded name 4.1.3, name 4.1, number declaration 3.2.2, object declaration 3.2.1, package 7, package declaration 7.1, package identifier 7.1, package specification 7.1, scope 8.2, simple name 4.1, type declaration 3.3.1, use clause 8.4, variable 3.2.1

7.3 Package Bodies

1   In contrast to the entities declared in the visible part of a package specification, the entities declared in the package body are only visible within the package body itself. As a consequence, a package with a package body can be used for the construction of a group of related subprograms (a *package* in the usual sense), in which the logical operations available to the users are clearly isolated from the internal entities.

2   For the elaboration of a package body, its declarative part is first elaborated,
     and its sequence of statements (if any) is then executed. The optional exception
     handlers at the end of a package body service exceptions raised during the
     execution of the sequence of statements of the package body.

     Notes:

3   A variable declared in the body of a package is only visible within this body
     and, consequently, its value can only be changed within the package body. In
     the absence of local tasks, the value of such a variable remains unchanged
     between calls issued from outside the package to subprograms declared in the
     visible part. The properties of such a variable are similar to those of an "own"
     variable of Algol 60.

4   The elaboration of the body of a subprogram declared in the visible part of
     a package is caused by the elaboration of the body of the package. Hence
     a call of such a subprogram by an outside program unit raises the exception
     PROGRAM_ERROR if the call takes place before the elaboration of the package
     body (see 3.9).

5   **Example of a package:**

```
package RATIONAL_NUMBERS is

   type RATIONAL is
      record
         NUMERATOR   : INTEGER;
         DENOMINATOR : POSITIVE;
      end record;

   function EQUAL(X,Y : RATIONAL) return BOOLEAN;

   function "/"  (X,Y : INTEGER)  return RATIONAL;
      --  to construct a rational number

   function "+"  (X,Y : RATIONAL) return RATIONAL;
   function "-"  (X,Y : RATIONAL) return RATIONAL;
   function "*"  (X,Y : RATIONAL) return RATIONAL;
   function "/"  (X,Y : RATIONAL) return RATIONAL;
end;

package body RATIONAL_NUMBERS is

   procedure SAME_DENOMINATOR (X,Y : in out RATIONAL) is
   begin
      --  reduces X and Y to the same denominator:
      ...
   end;
```

```
      function EQUAL(X,Y : RATIONAL) return BOOLEAN is
         U,V : RATIONAL;
      begin
         U := X;
         V := Y;
         SAME_DENOMINATOR (U,V);
         return U.NUMERATOR = V.NUMERATOR;
      end EQUAL;

      function "/" (X,Y : INTEGER) return RATIONAL is
      begin
         if Y > 0 then
            return (NUMERATOR => X,  DENOMINATOR => Y);
         else
            return (NUMERATOR => -X, DENOMINATOR => -Y);
         end if;
      end "/";

      function "+" (X,Y : RATIONAL) return RATIONAL is ...  end "+";
      function "-" (X,Y : RATIONAL) return RATIONAL is ...  end "-";
      function "*" (X,Y : RATIONAL) return RATIONAL is ...  end "*";
      function "/" (X,Y : RATIONAL) return RATIONAL is ...  end "/";

   end RATIONAL_NUMBERS;
```

6   **References:** declaration 3.1, declarative part 3.9, elaboration 3.1 3.9, exception 11, exception handler 11.2, name 4.1, package specification 7.1, program unit 6, program_ error exception 11.1, sequence of statements 5.1, subprogram 6, variable 3.2.1, visible part 7.2

7.4 Private Type and Deferred Constant Declarations

1   The declaration of a type as a private type in the visible part of a package serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself). Deferred constant declarations declare constants of private types.

2
```
      private_type_declaration ::=
         type identifier [discriminant_part] is [limited] private;

      deferred_constant_declaration ::=
         identifier_list : constant type_mark;
```

3   A private type declaration is only allowed as a declarative item of the visible part of a package, or as the generic parameter declaration for a generic formal type in a generic formal part.

4   The type mark of a deferred constant declaration must denote a private type or a subtype of a private type; a deferred constant declaration and the declaration of the corresponding private type must both be declarative items of the visible part of the same package. A deferred constant declaration with several identifiers is equivalent to a sequence of single deferred constant declarations as explained in section 3.2.

5   **Examples of private type declarations:**
```
type KEY is private;
type FILE_NAME is limited private;
```

6   **Example of deferred constant declaration:**
```
NULL_KEY : constant KEY;
```

7   **References:** constant 3.2.1, declaration 3.1, declarative item 3.9, deferred constant 7.4.3, discriminant part 3.7.1, generic formal part 12.1, generic formal type 12.1, generic parameter declaration 12.1, identifier 2.3, identifier list 3.2, limited type 7.4.4, package 7, private type 7.4.1, program unit 6, subtype 3.3, type 3.3, type mark 3.3.2, visible part 7.2

7.4.1 Private Types

1   If a private type declaration is given in the visible part of a package, then a corresponding declaration of a type with the same identifier must appear as a declarative item of the private part of the package. The corresponding declaration must be either a full type declaration or the declaration of a task type. In the rest of this section explanations are given in terms of full type declarations; the same rules apply also to declarations of task types.

2   A private type declaration and the corresponding full type declaration define a single type. The private type declaration, together with the visible part, define the operations that are available to outside program units (see section 7.4.2 on the operations that are available for private types). On the other hand, the full type declaration defines other operations whose direct use is only possible within the package itself.

3   If the private type declaration includes a discriminant part, the full declaration must include a discriminant part that conforms (see 6.3.1 for the conformance rules) and its type definition must be a record type definition. Conversely, if the private type declaration does not include a discriminant part, the type declared by the full type declaration (the *full type*) must not be an unconstrained type with discriminants. The full type must not be an unconstrained array type. A limited type (in particular a task type) is allowed for the full type only if the reserved word **limited** appears in the private type declaration (see 7.4.4).[1]

---

[1] See also Appendix G, AI-00232.

4   Within the specification of the package that declares a private type and before
the end of the corresponding full type declaration, a restriction applies to the
use of a name that denotes the private type or a subtype of the private type
and, likewise, to the use of a name that denotes any type or subtype that has
a subcomponent of the private type. The only allowed occurrences of such a
name are in a deferred constant declaration, a type or subtype declaration, a
subprogram specification, or an entry declaration; moreover, occurrences within
derived type definitions or within simple expressions are not allowed.[2]

5   The elaboration of a private type declaration creates a private type. If the
private type declaration has a discriminant part, this elaboration includes
that of the discriminant part. The elaboration of the full type declaration
consists of the elaboration of the type definition; the discriminant part, if any,
is not elaborated (since the conforming discriminant part of the private type
declaration has already been elaborated).

Notes:

6   It follows from the given rules that neither the declaration of a variable of
a private type, nor the creation by an allocator of an object of the private
type are allowed before the full declaration of the type. Similarly before the
full declaration, the name of the private type cannot be used in a generic
instantiation or in a representation clause.

7   **References:** allocator 4.8, array type 3.6, conform 6.3.1, declarative item 3.9, deferred
constant declaration 7.4.3, derived type 3.4, discriminant part 3.7.1, elaboration 3.9,
entry declaration 9.5, expression 4.4, full type declaration 3.3.1, generic instantiation
12.3, identifier 2.3, incomplete type declaration 3.8.1, limited type 7.4.4, name 4.1,
operation 3.3, package 7, package specification 7.1, private part 7.2, private type
7.4, private type declaration 7.4, record type definition 3.7, representation clause
13.1, reserved word 2.9, subcomponent 3.3, subprogram specification 6.1, subtype
3.3, subtype declaration 3.3.2, type 3.3, type declaration 3.3.1, type definition 3.3.1,
unconstrained array type 3.6, variable 3.2.1, visible part 7.2

7.4.2 Operations of a Private Type

1   The operations that are implicitly declared by a private type declaration
include basic operations. These are the operations involved in assignment
(unless the reserved word **limited** appears in the declaration), membership
tests, selected components for the selection of any discriminant, qualification,
and explicit conversions.

---

[2]   See also Appendix G, AI-00039, AI-00153, and AI-00384.

2 For a private type T, the basic operations also include the attributes T′BASE (see 3.3.3) and T′SIZE (see 13.7.2). For an object A of a private type, the basic operations include the attribute A′CONSTRAINED if the private type has discriminants (see 3.7.4), and in any case, the attributes A′SIZE and A′ADDRESS (see 13.7.2).

3 Finally, the operations implicitly declared by a private type declaration include the predefined comparison for equality and inequality unless the reserved word **limited** appears in the private type declaration.

4 The above operations, together with subprograms that have a parameter or result of the private type and that are declared in the visible part of the package, are the only operations from the package that are available outside the package for the private type.

5 Within the package that declares the private type, the additional operations implicitly declared by the full type declaration are also available. However, the redefinition of these implicitly declared operations is allowed within the same declarative region, including between the private type declaration and the corresponding full declaration. An explicitly declared subprogram hides an implicitly declared operation that has the same parameter and result type profile (this is only possible if the implicitly declared operation is a derived subprogram or a predefined operator).

6 If a composite type has subcomponents of a private type and is declared outside the package that declares the private type, then the operations that are implicitly declared by the declaration of the composite type include all operations that only depend on the characteristics that result from the private type declaration alone. (For example the operator < is not included for a one-dimensional array type.)

7 If the composite type is itself declared within the package that declares the private type (including within an inner package or generic package), then additional operations that depend on the characteristics of the full type are implicitly declared, as required by the rules applicable to the composite type (for example the operator < is declared for a one-dimensional array type if the full type is discrete). These additional operations are implicitly declared at the earliest place within the immediate scope of the composite type and after the full type declaration.[3]

8 The same rules apply to the operations that are implicitly declared for an access type whose designated type is a private type or a type declared by an incomplete type declaration.[4]

---

[3] See also Appendix G, AI-00139 and AI-00154.
[4] See also Appendix G, AI-00154.

9   For every private type or subtype T the following attribute is defined:[5]

10   T′CONSTRAINED   Yields the value FALSE if T denotes an unconstrained
                        nonformal private type with discriminants; also
                        yields the value FALSE if T denotes a generic formal
                        private type, and the associated actual subtype is
                        either an unconstrained type with discriminants
                        or an unconstrained array type; yields the value
                        TRUE otherwise. The value of this attribute is of the
                        predefined type BOOLEAN.

Note:

11   A private type declaration and the corresponding full type declaration define
     two different views of one and the same type. Outside of the defining package
     the characteristics of the type are those defined by the visible part. Within
     these outside program units the type is just a private type and any language
     rule that applies only to another class of types does not apply. The fact that the
     full declaration might *implement* the private type with a type of a particular
     class (for example, as an array type) is only relevant within the package itself.

12   The consequences of this actual implementation are, however, valid
     everywhere. For example: any default initialization of components takes
     place; the attribute SIZE provides the size of the full type; task dependence
     rules still apply to components that are task objects.

13   **Example:**

```
package KEY_MANAGER is
   type KEY is private;
   NULL_KEY : constant KEY;
   procedure GET_KEY(K : out KEY);
   function "<" (X, Y : KEY) return BOOLEAN;
private
   type KEY is new NATURAL;
   NULL_KEY : constant KEY := 0;
end;

package body KEY_MANAGER is
   LAST_KEY : KEY := 0;
   procedure GET_KEY(K : out KEY) is
   begin
      LAST_KEY := LAST_KEY + 1;
      K := LAST_KEY;
   end GET_KEY;
```

---

[5] See also Appendix G, AI-00026.

```
      function "<" (X, Y : KEY) return BOOLEAN is
      begin
         return INTEGER(X) < INTEGER(Y);
      end "<";
   end KEY_MANAGER;
```

Notes on the example:

14   Outside of the package KEY_MANAGER, the operations available for objects of type KEY include assignment, the comparison for equality or inequality, the procedure GET_KEY and the operator "<"; they do not include other relational operators such as ">=", or arithmetic operators.

15   The explicitly declared operator "<" hides the predefined operator "<" implicitly declared by the full type declaration. Within the body of the function, an explicit conversion of X and Y to the type INTEGER is necessary to invoke the "<" operator of this type. Alternatively, the result of the function could be written as **not** (X >= Y), since the operator ">=" is not redefined.

16   The value of the variable LAST_KEY, declared in the package body, remains unchanged between calls of the procedure GET_KEY. (See also the Notes of section 7.3.)

17   **References:** assignment 5.2, attribute 4.1.4, basic operation 3.3.3, component 3.3, composite type 3.3, conversion 4.6, declaration 3.1, declarative region 8.1, derived subprogram 3.4, derived type 3.4, dimension 3.6, discriminant 3.3, equality 4.5.2, full type 7.4.1, full type declaration 3.3.1, hiding 8.3, immediate scope 8.2, implicit declaration 3.1, incomplete type declaration 3.8.1, membership test 4.5, operation 3.3, package 7, parameter of a subprogram 6.2, predefined function 8.6, predefined operator 4.5, private type 7.4, private type declaration 7.4, program unit 6, qualification 4.7, relational operator 4.5, selected component 4.1.3, subprogram 6, task dependence 9.4, visible part 7.2

7.4.3 Deferred Constants

1   If a deferred constant declaration is given in the visible part of a package then a constant declaration (that is, an object declaration declaring a constant object, with an explicit initialization) with the same identifier must appear as a declarative item of the private part of the package. This object declaration is called the *full* declaration of the deferred constant. The type mark given in the full declaration must conform to that given in the deferred constant declaration (see 6.3.1). Multiple or single declarations are allowed for the deferred and the full declarations, provided that the equivalent single declarations conform.

2   Within the specification of the package that declares a deferred constant and before the end of the corresponding full declaration, the use of a name that denotes the deferred constant is only allowed in the default expression for a record component or for a formal parameter (not for a generic formal parameter).

3   The elaboration of a deferred constant declaration has no other effect.

4   The execution of a program is erroneous if it attempts to use the value of a deferred constant before the elaboration of the corresponding full declaration.[6]

Note:

5   The full declaration for a deferred constant that has a given private type must not appear before the corresponding full type declaration. This is a consequence of the rules defining the allowed uses of a name that denotes a private type (see 7.4.1).

6   **References:** conform 6.3.1, constant declaration 3.2.1, declarative item 3.9, default expression for a discriminant 3.7.1, deferred constant 7.4, deferred constant declaration 7.4, elaboration has no other effect 3.1, formal parameter 6.1, generic formal parameter 12.1 12.3, identifier 2.3, object declaration 3.2.1, package 7, package specification 7.1, private part 7.2, record component 3.7, type mark 3.3.2, visible part 7.2

7.4.4 Limited Types

1   A limited type is a type for which neither assignment nor the predefined comparison for equality and inequality is *implicitly* declared.

2   A private type declaration that includes the reserved word **limited** declares a limited type. A task type is a limited type. A type derived from a limited type is itself a limited type. Finally, a composite type is limited if the type of any of its subcomponents is limited.

3   The operations available for a private type that is limited are as given in section 7.4.2 for private types except for the absence of assignment and of a predefined comparison for equality and inequality.

4   For a formal parameter whose type is limited and whose declaration occurs in an explicit subprogram declaration, the mode **out** is only allowed if this type is private and the subprogram declaration occurs within the visible part of the package that declares the private type. The same holds for formal parameters of entry declarations and of generic procedure declarations. The corresponding full type must not be limited if the mode **out** is used for any such formal parameter. Otherwise, the corresponding full type is allowed (but not required) to be a limited type (in particular, it is allowed to be a task type). If the full type corresponding to a limited private type is not itself limited, then assignment for the type is available within the package, but not outside.[7]

5   The following are consequences of the rules for limited types:

---

[6] See also Appendix G, AI-00155.
[7] See also Appendix G, AI-00260.

6 • An explicit initialization is not allowed in an object declaration if the type of the object is limited.

7 • A default expression is not allowed in a component declaration if the type of the record component is limited.

8 • An explicit initial value is not allowed in an allocator if the designated type is limited.

9 • A generic formal parameter of mode **in** must not be of a limited type.

Notes:

10 The above rules do not exclude a default expression for a formal parameter of a limited type; they do not exclude a deferred constant of a limited type if the full type is not limited. An explicit declaration of an equality operator is allowed for a limited type (see 6.7).

11 Aggregates are not available for a limited composite type (see 3.6.2 and 3.7.4). Catenation is not available for a limited array type (see 3.6.2).

12 **Example:**

```
package I_O_PACKAGE is
   type FILE_NAME is limited private;

   procedure OPEN (F : in out FILE_NAME);
   procedure CLOSE(F : in out FILE_NAME);
   procedure READ (F : in FILE_NAME; ITEM : out INTEGER);
   procedure WRITE(F : in FILE_NAME; ITEM : in  INTEGER);
private
   type FILE_NAME is
      record
         INTERNAL_NAME : INTEGER := 0;
      end record;
end I_O_PACKAGE;

package body I_O_PACKAGE is
   LIMIT : constant := 200;
   type FILE_DESCRIPTOR is record  ...  end record;
   DIRECTORY : array (1 .. LIMIT) of FILE_DESCRIPTOR;
   ...
   procedure OPEN (F    : in out FILE_NAME) is  ...  end;
   procedure CLOSE(F    : in out FILE_NAME) is  ...  end;
   procedure READ (F    : in FILE_NAME;
                   ITEM : out INTEGER) is ... end;
   procedure WRITE(F    : in FILE_NAME;
                   ITEM : in  INTEGER) is ... end;
begin
   ...
end I_O_PACKAGE;
```

Notes on the example:

13   In the example above, an outside subprogram making use of I_O_PACKAGE may obtain a file name by calling OPEN and later use it in calls to READ and WRITE. Thus, outside the package, a file name obtained from OPEN acts as a kind of password; its internal properties (such as containing a numeric value) are not known and no other operations (such as addition or comparison of internal names) can be performed on a file name.

14   This example is characteristic of any case where complete control over the operations of a type is desired. Such packages serve a dual purpose. They prevent a user from making use of the internal structure of the type. They also implement the notion of an *encapsulated* data type where the only operations on the type are those given in the package specification.

15   **References:** aggregate 4.3, allocator 4.8, assignment 5.2, catenation operator 4.5, component declaration 3.7, component type 3.3, composite type 3.3, default expression for a discriminant 3.7, deferred constant 7.4.3, derived type 3.4, designate 3.8, discriminant specification 3.7.1, equality 4.5.2, formal parameter 6.1, full type 7.4.1, full type declaration 3.3.1, generic formal parameter 12.1 12.3, implicit declaration 3.1, initial value 3.2.1, mode 12.1.1, object 3.2, operation 3.3, package 7, predefined operator 4.5, private type 7.4, private type declaration 7.4, record component 3.7, record type 3.7, relational operator 4.5, subcomponent 3.3, subprogram 6, task type 9.1 9.2, type 3.3

7.5 Example of a Table Management Package

1   The following example illustrates the use of packages in providing high level procedures with a simple interface to the user.

2   The problem is to define a table management package for inserting and retrieving items. The items are inserted into the table as they are supplied. Each inserted item has an order number. The items are retrieved according to their order number, where the item with the lowest order number is retrieved first.

3   From the user's point of view, the package is quite simple. There is a type called ITEM designating table items, a procedure INSERT for inserting items, and a procedure RETRIEVE for obtaining the item with the lowest order number. There is a special item NULL_ITEM that is returned when the table is empty, and an exception TABLE_FULL which is raised by INSERT if the table is already full.

4   A sketch of such a package is given below. Only the specification of the package
is exposed to the user.

5

```
package TABLE_MANAGER is

    type ITEM is
       record
          ORDER_NUM : INTEGER;
          ITEM_CODE : INTEGER;
          QUANTITY  : INTEGER;
          ITEM_TYPE : CHARACTER;
       end record;

    NULL_ITEM : constant ITEM :=
       (ORDER_NUM | ITEM_CODE | QUANTITY => 0, ITEM_TYPE => ' ');

    procedure INSERT  (NEW_ITEM   : in  ITEM);
    procedure RETRIEVE(FIRST_ITEM : out ITEM);

    TABLE_FULL : exception;  --  raised by INSERT when table full
end;
```

6   The details of implementing such packages can be quite complex; in this case
they involve a two-way linked table of internal items. A local housekeeping
procedure EXCHANGE is used to move an internal item between the busy and
the free lists. The initial table linkages are established by the initialization
part. The package body need not be shown to the users of the package.

7

```
package body TABLE_MANAGER is
    SIZE : constant := 2000;
    subtype INDEX is INTEGER range 0 .. SIZE;

    type INTERNAL_ITEM is
       record
          CONTENT : ITEM;
          SUCC    : INDEX;
          PRED    : INDEX;
       end record;

    TABLE : array (INDEX) of INTERNAL_ITEM;
    FIRST_BUSY_ITEM : INDEX := 0;
    FIRST_FREE_ITEM : INDEX := 1;

    function FREE_LIST_EMPTY return BOOLEAN is ... end;
    function BUSY_LIST_EMPTY return BOOLEAN is ... end;
    procedure EXCHANGE (FROM : in INDEX;
                          TO : in INDEX) is ... end;
```

```
      procedure INSERT (NEW_ITEM : in ITEM) is
      begin
         if FREE_LIST_EMPTY then
            raise TABLE_FULL;
         end if;
         -- remaining code for INSERT
      end INSERT;

      procedure RETRIEVE (FIRST_ITEM : out ITEM) is ... end;

   begin
      -- initialization of the table linkages
   end TABLE_MANAGER;
```

7.6 Example of a Text Handling Package

1 This example illustrates a simple text handling package. The users only have access to the visible part; the implementation is hidden from them in the private part and the package body (not shown).

2 From a user's point of view, a TEXT is a variable-length string. Each text object has a maximum length, which must be given when the object is declared, and a current value, which is a string of some length between zero and the maximum. The maximum possible length of a text object is an implementation-defined constant.

3 The package defines first the necessary types, then functions that return some characteristics of objects of the type, then the conversion functions between texts and the predefined CHARACTER and STRING types, and finally some of the standard operations on varying strings. Most operations are overloaded on strings and characters as well as on the type TEXT, in order to minimize the number of explicit conversions the user has to write.

4
```
   package TEXT_HANDLER is
      MAXIMUM : constant := SOME_VALUE;  -- implementation-defined
      subtype INDEX is INTEGER range 0 .. MAXIMUM;

      type TEXT(MAXIMUM_LENGTH : INDEX) is limited private;

      function LENGTH (T : TEXT) return INDEX;
      function VALUE  (T : TEXT) return STRING;
      function EMPTY  (T : TEXT) return BOOLEAN;

      function TO_TEXT (S : STRING;
                        MAX : INDEX)   return TEXT;
                        -- maximum length MAX
      function TO_TEXT (C : CHARACTER;
                        MAX : INDEX)   return TEXT;
      function TO_TEXT (S : STRING)    return TEXT;
                        -- maximum length S'LENGTH
      function TO_TEXT (C : CHARACTER) return TEXT;
```

```
      function "&" (LEFT  : TEXT;
                    RIGHT : TEXT)       return TEXT;
      function "&" (LEFT  : TEXT;
                    RIGHT : STRING)     return TEXT;
      function "&" (LEFT  : STRING;
                    RIGHT : TEXT)       return TEXT;
      function "&" (LEFT  : TEXT;
                    RIGHT : CHARACTER)  return TEXT;
      function "&" (LEFT  : CHARACTER;
                    RIGHT : TEXT)       return TEXT;

      function "="  (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
      function "<"  (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
      function "<=" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
      function ">"  (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
      function ">=" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;

      procedure SET (OBJECT : in out TEXT; VALUE : in TEXT);
      procedure SET (OBJECT : in out TEXT; VALUE : in STRING);
      procedure SET (OBJECT : in out TEXT; VALUE : in CHARACTER);

      procedure APPEND (TAIL : in TEXT;      TO : in out TEXT);
      procedure APPEND (TAIL : in STRING;    TO : in out TEXT);
      procedure APPEND (TAIL : in CHARACTER; TO : in out TEXT);

      procedure AMEND (OBJECT   : in out TEXT; BY : in TEXT;
                       POSITION : in INDEX);
      procedure AMEND (OBJECT   : in out TEXT; BY : in STRING;
                       POSITION : in INDEX);
      procedure AMEND (OBJECT   : in out TEXT; BY : in CHARACTER;
                       POSITION : in INDEX);

      --  amend replaces part of the object by the given
      --  text, string, or character
      --  starting at the given position in the object

      function LOCATE (FRAGMENT : TEXT;
                       WITHIN   : TEXT) return INDEX;
      function LOCATE (FRAGMENT : STRING;
                       WITHIN   : TEXT) return INDEX;
      function LOCATE (FRAGMENT : CHARACTER;
                       WITHIN   : TEXT) return INDEX;

      --  all return 0 if the fragment is not located
   private
      type TEXT(MAXIMUM_LENGTH : INDEX) is
         record
            POS   : INDEX := 0;
            VALUE : STRING(1 .. MAXIMUM_LENGTH);
         end record;
   end TEXT_HANDLER;
```

5 **Example of use of the text handling package:**

6 A program opens an output file, whose name is supplied by the string NAME. This string has the form

```
[DEVICE :] [FILENAME [.EXTENSION]]
```

7 There are standard defaults for device, filename, and extension. The user-supplied name is passed to EXPAND_FILE_NAME as a parameter, and the result is the expanded version, with any necessary defaults added.

8
```
function EXPAND_FILE_NAME (NAME : STRING) return STRING is
    use TEXT_HANDLER;

    DEFAULT_DEVICE     : constant STRING := "SY:";
    DEFAULT_FILE_NAME  : constant STRING := "RESULTS";
    DEFAULT_EXTENSION  : constant STRING := ".DAT";

    MAXIMUM_FILE_NAME_LENGTH :
                       constant INDEX := SOME_APPROPRIATE_VALUE;
    FILE_NAME : TEXT(MAXIMUM_FILE_NAME_LENGTH);

begin

    SET(FILE_NAME, NAME);

    if EMPTY(FILE_NAME) then
        SET(FILE_NAME, DEFAULT_FILE_NAME);
    end if;

    if LOCATE(':', FILE_NAME) = 0 then
        SET(FILE_NAME, DEFAULT_DEVICE & FILE_NAME);
    end if;

    if LOCATE('.', FILE_NAME) = 0 then
        APPEND(DEFAULT_EXTENSION, TO => FILE_NAME);
    end if;

    return VALUE(FILE_NAME);

end EXPAND_FILE_NAME;
```

8

Visibility Rules

1 The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the program are described in this chapter. The formulation of these rules uses the notion of a declarative region.

2 **References:** declaration 3.1, declarative region 8.1, identifier 2.3, scope 8.2, visibility 8.3

8.1 Declarative Region

1 A declarative region is a portion of the program text. A single declarative region is formed by the text of each of the following:

2 • A subprogram declaration, a package declaration, a task declaration, or a generic declaration, together with the corresponding body, if any. If the body is a body stub, the declarative region also includes the corresponding subunit. If the program unit has subunits, they are also included.

3 • An entry declaration together with the corresponding accept statements.

4 • A record type declaration, together with a corresponding private or incomplete type declaration if any, and together with a corresponding record representation clause if any.

5 • A renaming declaration that includes a formal part, or a generic parameter declaration that includes either a formal part or a discriminant part.

6 • A block statement or a loop statement.

7 In each of the above cases, the declarative region is said to be *associated* with the corresponding declaration or statement. A declaration is said to *occur immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself.

8   A declaration that occurs immediately within a declarative region is said to be *local* to the region. Declarations in outer (enclosing) regions are said to be *global* to an inner (enclosed) declarative region. A local entity is one declared by a local declaration; a global entity is one declared by a global declaration.

9   Some of the above forms of declarative region include several disjoint parts (for example, other declarative items can be between the declaration of a package and its body). Each declarative region is nevertheless considered as a (logically) continuous portion of the program text. Hence if any rule defines a portion of text as the text that *extends* from some specific point of a declarative region to the end of this region, then this portion is the corresponding subset of the declarative region (for example it does not include intermediate declarative items between the two parts of a package).

Notes:

10   As defined in section 3.1, the term declaration includes basic declarations, implicit declarations, and those declarations that are part of basic declarations, for example, discriminant and parameter specifications. It follows from the definition of a declarative region that a discriminant specification occurs immediately within the region associated with the enclosing record type declaration. Similarly, a parameter specification occurs immediately within the region associated with the enclosing subprogram body or accept statement.

11   The package STANDARD forms a declarative region which encloses all library units: the implicit declaration of each library unit is assumed to occur immediately within this package (see sections 8.6 and 10.1.1).

12   Declarative regions can be nested within other declarative regions. For example, subprograms, packages, task units, generic units, and block statements can be nested within each other, and can contain record type declarations, loop statements, and accept statements.

13   **References:** accept statement 9.5, basic declaration 3.1, block statement 5.6, body stub 10.2, declaration 3.1, discriminant part 3.7.1, discriminant specification 3.7.1, entry declaration 9.5, formal part 6.1, generic body 12.2, generic declaration 12.1, generic parameter declaration 12.1, implicit declaration 3.1, incomplete type declaration 3.8.1, library unit 10.1, loop statement 5.5, package 7, package body 7.1, package declaration 7.1, parameter specification 6.1, private type declaration 7.4, record representation clause 13.4, record type 3.7, renaming declaration 8.5, standard package 8.6, subprogram body 6.3, subprogram declaration 6.1, subunit 10.2, task body 9.1, task declaration 9.1, task unit 9

8.2 Scope of Declarations

1   For each form of declaration, the language rules define a certain portion of the program text called the *scope* of the declaration. The scope of a declaration is also called the scope of any entity declared by the declaration. Furthermore, if the declaration associates some notation with a declared entity, this portion of the text is also called the scope of this notation (either an identifier, a character literal, an operator symbol, or the notation for a basic operation). Within the scope of an entity, and only there, there are places where it is legal to use the associated notation in order to refer to the declared entity. These places are defined by the rules of visibility and overloading.

2   The scope of a declaration that occurs immediately within a declarative region extends from the beginning of the declaration to the end of the declarative region; this part of the scope of a declaration is called the *immediate scope.* Furthermore, for any of the declarations listed below, the scope of the declaration extends beyond the immediate scope:

3   (a)   A declaration that occurs immediately within the visible part of a package declaration.

4   (b)   An entry declaration.

5   (c)   A component declaration.

6   (d)   A discriminant specification.

7   (e)   A parameter specification.

8   (f)   A generic parameter declaration.

9   In each of these cases, the given declaration occurs immediately within some enclosing declaration, and the scope of the given declaration extends to the end of the scope of the enclosing declaration.

10   In the absence of a subprogram declaration, the subprogram specification given in the subprogram body or in the body stub acts as the declaration and rule (e) applies also in such a case.

Note:

11   The above scope rules apply to all forms of declaration defined by section 3.1; in particular, they apply also to implicit declarations. Rule (a) applies to a package declaration and thus not to the package specification of a generic declaration. For nested declarations, the rules (a) through (f) apply at each level. For example, if a task unit is declared in the visible part of a package, the scope of an entry of the task unit extends to the end of the scope of the task unit, that is, to the end of the scope of the enclosing package. The scope of a use clause is defined in section 8.4.

₁₂ **References:** basic operation 3.3.3, body stub 10.2, character literal 2.5, component declaration 3.7, declaration 3.1, declarative region 8.1, discriminant specification 3.7.1, entry declaration 9.5, extends 8.1, generic declaration 12.1, generic parameter declaration 12.1, identifier 2.3, implicit declaration 3.1, occur immediately within 8.1, operator symbol 6.1, overloading 6.6 8.7, package declaration 7.1, package specification 7.1, parameter specification 6.1, record type 3.7, renaming declaration 8.5, subprogram body 6.3, subprogram declaration 6.1, task declaration 9.1, task unit 9, type declaration 3.3.1, use clause 8.4, visibility 8.3, visible part 7.2

8.3 Visibility

₁ The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the case of overloaded declarations, by the overloading rules. The identifiers considered in this chapter include any identifier other than a reserved word, an attribute designator, a pragma identifier, the identifier of a pragma argument, or an identifier given as a pragma argument. The places considered in this chapter are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this chapter are those for subprograms, enumeration literals, and single entries.

₂ For each identifier and at each place in the text, the visibility rules determine a set of declarations (with this identifier) that define possible meanings of an occurrence of the identifier. A declaration is said to be *visible* at a given place in the text when, according to the visibility rules, the declaration defines a possible meaning of this occurrence. Two cases arise.

₃ • The visibility rules determine *at most one* possible meaning. In such a case the visibility rules are sufficient to determine the declaration defining the meaning of the occurrence of the identifier, or in the absence of such a declaration, to determine that the occurrence is not legal at the given point.

₄ • The visibility rules determine *more than one* possible meaning. In such a case the occurrence of the identifier is legal at this point if and only if *exactly one* visible declaration is acceptable for the overloading rules in the given context (see section 6.6 for the rules of overloading and section 8.7 for the context used for overload resolution).

₅ A declaration is only visible within a certain part of its scope; this part starts at the end of the declaration except in a package specification, in which case it starts at the reserved word **is** given after the identifier of the package specification. (This rule applies, in particular, for implicit declarations.)

₆ Visibility is either by selection or direct. A declaration is visible *by selection* at places that are defined as follows.

7   (a)   For a declaration given in the visible part of a package declaration: at the place of the selector after the dot of an expanded name whose prefix denotes the package.

8   (b)   For an entry declaration of a given task type: at the place of the selector after the dot of a selected component whose prefix is appropriate for the task type.

9   (c)   For a component declaration of a given record type declaration: at the place of the selector after the dot of a selected component whose prefix is appropriate for the type; also at the place of a component simple name (before the compound delimiter =>) in a named component association of an aggregate of the type.

10   (d)   For a discriminant specification of a given type declaration: at the same places as for a component declaration; also at the place of a discriminant simple name (before the compound delimiter =>) in a named discriminant association of a discriminant constraint for the type.

11   (e)   For a parameter specification of a given subprogram specification or entry declaration: at the place of the formal parameter (before the compound delimiter =>) in a named parameter association of a corresponding subprogram or entry call.

12   (f)   For a generic parameter declaration of a given generic unit: at the place of the generic formal parameter (before the compound delimiter =>) in a named generic association of a corresponding generic instantiation.

13   Finally, within the declarative region associated with a construct other than a record type declaration, any declaration that occurs immediately within the region is visible by selection at the place of the selector after the dot of an expanded name whose prefix denotes the construct.

14   Where it is not visible by selection, a visible declaration is said to be *directly visible.* A declaration is directly visible within a certain part of its immediate scope; this part extends to the end of the immediate scope of the declaration, but excludes places where the declaration is hidden as explained below. In addition, a declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause according to the rules described in section 8.4. (See also section 8.6 for the visibility of library units.)

15 A declaration is said to be *hidden* within (part of) an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden within the immediate scope of the inner homograph. Each of two declarations is said to be a *homograph* of the other if both declarations have the same identifier and overloading is allowed for at most one of the two. If overloading is allowed for both declarations, then each of the two is a homograph of the other if they have the same identifier, operator symbol, or character literal, as well as the same parameter and result type profile (see 6.6).[1]

16 Within the specification of a subprogram, every declaration with the same designator as the subprogram is hidden; the same holds within a generic instantiation that declares a subprogram, and within an entry declaration or the formal part of an accept statement; where hidden in this manner, a declaration is visible neither by selection nor directly.[2]

17 Two declarations that occur immediately within the same declarative region must not be homographs, unless either or both of the following requirements are met: (a) exactly one of them is the implicit declaration of a predefined operation; (b) exactly one of them is the implicit declaration of a derived subprogram. In such cases, a predefined operation is always hidden by the other homograph; a derived subprogram hides a predefined operation, but is hidden by any other homograph. Where hidden in this manner, an implicit declaration is hidden within the entire scope of the other declaration (regardless of which declaration occurs first); the implicit declaration is visible neither by selection nor directly.[3]

18 Whenever a declaration with a certain identifier is visible from a given point, the identifier and the declared entity (if any) are also said to be visible from that point. Direct visibility and visibility by selection are likewise defined for character literals and operator symbols. An operator is directly visible if and only if the corresponding operator declaration is directly visible. Finally, the notation associated with a basic operation is directly visible within the entire scope of this operation.[4]

---

[1] See also Appendix G, AI-00286.
[2] See also Appendix G, AI-00370.
[3] See also Appendix G, AI-00002, AI-00012, and AI-00330.
[4] See also Appendix G, AI-00027.

₁₉ **Example:**
```
procedure P is
   A, B : BOOLEAN;

   procedure Q is
      C : BOOLEAN;
      B : BOOLEAN;  --  an inner homograph of B
   begin
      ...
      B := A;    --  means Q.B := P.A;
      C := P.B;  --  means Q.C := P.B;
   end;
begin
   ...
   A := B;  --  means P.A := P.B;
end;
```

Note on the visibility of library units:

₂₀ The visibility of library units is determined by with clauses (see 10.1.1) and by the fact that library units are implicitly declared in the package STANDARD (see 8.6).

Note on homographs:

₂₁ The same identifier may occur in different declarations and may thus be associated with different entities, even if the scopes of these declarations overlap. Overlap of the scopes of declarations with the same identifier can result from overloading of subprograms and of enumeration literals. Such overlaps can also occur for entities declared in package visible parts and for entries, record components, and parameters, where there is overlap of the scopes of the enclosing package declarations, task declarations, record type declarations, subprogram declarations, renaming declarations, or generic declarations. Finally overlapping scopes can result from nesting.

Note on immediate scope, hiding, and visibility:

₂₂ The rules defining immediate scope, hiding, and visibility imply that a reference to an identifier within its own declaration is illegal (except for packages and generic packages). The identifier hides outer homographs within its immediate scope, that is, from the start of the declaration; on the other hand, the identifier is visible only after the end of the declaration. For this reason, all but the last of the following declarations are illegal:

```
K : INTEGER := K * K;         -- illegal
T : T;                        -- illegal
procedure P(X : P);           -- illegal

procedure Q(X : REAL := Q);   -- illegal, even if there is
                              -- a function named Q
```

```
procedure R(R : REAL);              --  an inner declaration
                                    --  is legal (although
                                    --  confusing)
```

23   **References:** accept statement 9.5, aggregate 4.3, appropriate for a type 4.1, argument
       2.8, basic operation 3.3.3, character literal 2.5, component association 4.3, component
       declaration 3.7, compound delimiter 2.2, declaration 3.1, declarative region 8.1,
       designate 3.8, discriminant constraint 3.7.2, discriminant specification 3.7.1, entry
       call 9.5, entry declaration 9.5, entry family 9.5, enumeration literal specification
       3.5.1, expanded name 4.1.3, extends 8.1, formal parameter 6.1, generic association
       12.3, generic formal parameter 12.1, generic instantiation 12.3, generic package 12.1,
       generic parameter declaration 12.1, generic unit 12, identifier 2.3, immediate scope
       8.2, implicit declaration 3.1, lexical element 2.2, library unit 10.1, object 3.2, occur
       immediately within 8.1, operator 4.5, operator symbol 6.1, overloading 6.6 8.7, package
       7, parameter 6.2, parameter association 6.4, parameter specification 6.1, pragma 2.8,
       program unit 6, record type 3.7, reserved word 2.9, scope 8.2, selected component
       4.1.3, selector 4.1.3, simple name 4.1, subprogram 6, subprogram call 6.4, subprogram
       declaration 6.1, subprogram specification 6.1, task type 9.1, task unit 9, type 3.3, type
       declaration 3.3.1, use clause 8.4, visible part 7.2

8.4 Use Clauses

1   A use clause achieves direct visibility of declarations that appear in the visible
       parts of named packages.

2   use_clause ::= **use** *package*_name { , *package*_name};

3   For each use clause, there is a certain region of text called the *scope* of the use
       clause. This region starts immediately after the use clause. If a use clause is
       a declarative item of some declarative region, the scope of the clause extends
       to the end of the declarative region. If a use clause occurs within a context
       clause of a compilation unit, the scope of the use clause extends to the end of
       the declarative region associated with the compilation unit.

4   In order to define which declarations are made directly visible at a given place
       by use clauses, consider the set of packages named by all use clauses whose
       scopes enclose this place, omitting from this set any packages that enclose
       this place. A declaration that can be made directly visible by a use clause
       (a potentially visible declaration) is any declaration that occurs immediately
       within the visible part of a package of the set. A potentially visible declaration
       is actually made directly visible except in the following two cases:

5   • A potentially visible declaration is not made directly visible if the
             place considered is within the immediate scope of a homograph of the
             declaration.[5]

---

[5]   See also Appendix G, AI-00286.

6 •   Potentially visible declarations that have the same identifier
       are not made directly visible unless each of them is either an
       enumeration literal specification or the declaration of a subprogram
       (by a subprogram declaration, a renaming declaration, a generic
       instantiation, or an implicit declaration).

7   The elaboration of a use clause has no other effect.

    Note:

8   The above rules guarantee that a declaration that is made directly visible by
    a use clause cannot hide an otherwise directly visible declaration. The above
    rules are formulated in terms of the set of packages named by use clauses.

9   Consequently, the following lines of text all have the same effect (assuming
    only one package P).

```
use P;
use P; use P, P;
```

10 **Example of conflicting names in two packages:**

```
procedure R is
   package TRAFFIC is
      type COLOR is (RED, AMBER, GREEN);
      ...
   end TRAFFIC;

   package WATER_COLORS is
      type COLOR is (WHITE, RED, YELLOW, GREEN,
                     BLUE, BROWN, BLACK);
      ...
   end WATER_COLORS;

   use TRAFFIC;        --  COLOR, RED, AMBER, and GREEN
                       --  are directly visible

   use WATER_COLORS;  --  two homographs of GREEN
                       --  are directly visible but
                       --  COLOR is no longer directly visible

   subtype LIGHT is TRAFFIC.COLOR;       -- Subtypes are used
   subtype SHADE is WATER_COLORS.COLOR; -- to resolve the
                                         -- conflicting
                                         -- type name COLOR

   SIGNAL : LIGHT;
   PAINT  : SHADE;
begin
   SIGNAL := GREEN;  --  that of TRAFFIC
   PAINT  := GREEN;  --  that of WATER_COLORS
end R;
```

₁₁ **Example of name identification with a use clause:**

```
package D is
   T, U, V : BOOLEAN;
end D;

procedure P is
   package E is
      B, W, V : INTEGER;
   end E;

   procedure Q is
      T, X : REAL;
      use D, E;
   begin
      --  the name T means Q.T, not D.T
      --  the name U means D.U
      --  the name B means E.B
      --  the name W means E.W
      --  the name X means Q.X
      --  the name V is illegal : either
      --  D.V or E.V must be used
      ...
   end Q;
begin
   ...
end P;
```

₁₂ **References:** compilation unit 10.1, context clause 10.1, declaration 3.1, declarative item 3.9, declarative region 8.1, direct visibility 8.3, elaboration 3.1 3.9, elaboration has no other effect 3.1, enumeration literal specification 3.5.1, extends 8.1, hiding 8.3, homograph 8.3, identifier 2.3, immediate scope 8.2, name 4.1, occur immediately within 8.1, package 7, scope 8.2, subprogram declaration 6.1, visible part 7.2

8.5 Renaming Declarations

₁ A renaming declaration declares another name for an entity.

```
2        renaming_declaration ::=
             identifier : type_mark    renames object_name;
           | identifier : exception    renames exception_name;
           | package identifier        renames package_name;
           | subprogram_specification renames subprogram_or_entry_name;
```

₃ The elaboration of a renaming declaration evaluates the name that follows the reserved word **renames** and thereby determines the entity denoted by this name (the renamed entity). At any point where a renaming declaration is visible, the identifier, or operator symbol of this declaration denotes the renamed entity.

4   The first form of renaming declaration is used for the renaming of objects. The renamed entity must be an object of the base type of the type mark. The properties of the renamed object are not affected by the renaming declaration. In particular, its value and whether or not it is a constant are unaffected; similarly, the constraints that apply to an object are not affected by renaming (any constraint implied by the type mark of the renaming declaration is ignored). The renaming declaration is legal only if exactly one object has this type and can be denoted by the object name.[6]

5   The following restrictions apply to the renaming of a subcomponent that depends on discriminants of a variable. The renaming is not allowed if the subtype of the variable, as defined in a corresponding object declaration, component declaration, or component subtype indication, is an unconstrained type; or if the variable is a generic formal object (of mode **in out**). Similarly if the variable is a formal parameter, the renaming is not allowed if the type mark given in the parameter specification denotes an unconstrained type whose discriminants have default expressions.[7]

6   The second form of renaming declaration is used for the renaming of exceptions; the third form, for the renaming of packages.

7   The last form of renaming declaration is used for the renaming of subprograms and entries. The renamed subprogram or entry and the subprogram specification given in the renaming declaration must have the same parameter and result type profile (see 6.6). The renaming declaration is legal only if exactly one visible subprogram or entry satisfies the above requirements and can be denoted by the given subprogram or entry name. In addition, parameter modes must be identical for formal parameters that are at the same parameter position.

8   The subtypes of the parameters and result (if any) of a renamed subprogram or entry are not affected by renaming. These subtypes are those given in the original subprogram declaration, generic instantiation, or entry declaration (not those of the renaming declaration); even for calls that use the new name. On the other hand, a renaming declaration can introduce parameter names and default expressions that differ from those of the renamed subprogram; named associations of calls with the new subprogram name must use the new parameter name; calls with the old subprogram name must use the old parameter names.[8]

---

[6] See also Appendix G, AI-00001.
[7] See also Appendix G, AI-00170 and AI-00502.
[8] See also Appendix G, AI-00245.

9   A procedure can only be renamed as a procedure. Either of a function or
      operator can be renamed as either of a function or operator; for renaming as
      an operator, the subprogram specification given in the renaming declaration is
      subject to the rules given in section 6.7 for operator declarations. Enumeration
      literals can be renamed as functions; similarly, attributes defined as functions
      (such as SUCC and PRED) can be renamed as functions. An entry can only be
      renamed as a procedure; the new name is only allowed to appear in contexts
      that allow a procedure name. An entry of a family can be renamed, but an
      entry family cannot be renamed as a whole.

10   **Examples:**
```
declare
    L : PERSON renames LEFTMOST_PERSON; -- see 3.8.1
begin
    L.AGE := L.AGE + 1;
end;

FULL : exception renames TABLE_MANAGER.TABLE_FULL; -- see 7.5

package TM renames TABLE_MANAGER;

function REAL_PLUS(LEFT, RIGHT : REAL   )
    return REAL     renames "+";
function INT_PLUS (LEFT, RIGHT : INTEGER)
    return INTEGER renames "+";

function ROUGE return COLOR renames RED;  --  see 3.5.1
function ROT   return COLOR renames RED;
function ROSSO return COLOR renames ROUGE;

function NEXT(X : COLOR) return COLOR renames COLOR'SUCC;
    -- see 3.5.5
```

11   **Example of a renaming declaration with new parameter names:**
```
function "*" (X,Y : VECTOR) return REAL renames DOT_PRODUCT;
    -- see 6.1
```

12   **Example of a renaming declaration with a new default expression:**
```
function MINIMUM(L : LINK := HEAD) return CELL renames MIN_CELL;
    -- see 6.1
```

      Notes:

13   Renaming may be used to resolve name conflicts and to act as a shorthand.
      Renaming with a different identifier or operator symbol does not hide the old
      name; the new name and the old name need not be visible at the same points.
      The attributes POS and VAL cannot be renamed since the corresponding
      specifications cannot be written; the same holds for the predefined multiplying
      operators with a *universal_fixed* result.

14　Calls with the new name of a renamed entry are procedure call statements and are not allowed at places where the syntax requires an entry call statement in conditional and timed entry calls; similarly, the COUNT attribute is not available for the new name.

15　A task object that is declared by an object declaration can be renamed as an object. However, a single task cannot be renamed since the corresponding task type is anonymous. For similar reasons, an object of an anonymous array type cannot be renamed. No syntactic form exists for renaming a generic unit.

16　A subtype can be used to achieve the effect of renaming a type (including a task type) as in

subtype MODE **is** TEXT_IO.FILE_MODE;

17　**References:** allow 1.6, attribute 4.1.4, base type 3.3, conditional entry call 9.7.2, constant 3.2.1, constrained subtype 3.3, constraint 3.3, declaration 3.1, default expression 6.1, depend on a discriminant 3.7.1, discriminant 3.7.1, elaboration 3.1 3.9, entry 9.5, entry call 9.5, entry call statement 9.5, entry declaration 9.5, entry family 9.5, enumeration literal 3.5.1, evaluation of a name 4.1, exception 11, formal parameter 6.1, function 6.5, identifier 2.3, legal 1.6, mode 6.1, name 4.1, object 3.2, object declaration 3.2, operator 6.7, operator declaration 6.7, operator symbol 6.1, package 7, parameter 6.2, parameter specification 6.1, procedure 6.1, procedure call statement 6.4, reserved word 2.9, subcomponent 3.3, subprogram 6, subprogram call 6.4, subprogram declaration 6.1, subprogram specification 6.1, subtype 3.3.2, task object 9.2, timed entry call 9.7.3, type 3.3, type mark 3.3.2, variable 3.2.1, visibility 8.3

8.6　The Package Standard

1　The predefined types (for example the types BOOLEAN, CHARACTER and INTEGER) are the types that are declared in a predefined package called STANDARD; this package also includes the declarations of their predefined operations. The package STANDARD is described in Annex C. Apart from the predefined numeric types, the specification of the package STANDARD must be the same for all implementations of the language.

2　The package STANDARD forms a declarative region which encloses every library unit and consequently the main program; the declaration of every library unit is assumed to occur immediately within this package. The implicit declarations of library units are assumed to be ordered in such a way that the scope of a given library unit includes any compilation unit that mentions the given library unit in a with clause. However, the only library units that are visible within a given compilation unit are as follows: they include the library units named by all with clauses that apply to the given unit, and moreover, if

the given unit is a secondary unit of some library unit, they include this library unit.[9]

Notes:

3 If all block statements of a program are named, then the name of each program unit can always be written as an expanded name starting with STANDARD (unless this package is itself hidden).

4 If a type is declared in the visible part of a library package, then it is a consequence of the visibility rules that a basic operation (such as assignment) for this type is directly visible at places where the type itself is not visible (whether by selection or directly). However this operation can only be applied to operands that are visible and the declaration of these operands requires the visibility of either the type or one of its subtypes.

5 **References:** applicable with clause 10.1.1, block name 5.6, block statement 5.6, declaration 3.1, declarative region 8.1, expanded name 4.1.3, hiding 8.3, identifier 2.3, implicit declaration 3.1, library unit 10.1, loop statement 5.5, main program 10.1, must 1.6, name 4.1, occur immediately within 8.1, operator 6.7, package 7, program unit 6, secondary unit 10.1, subtype 3.3, type 3.3, visibility 8.3, with clause 10.1.1

8.7 The Context of Overload Resolution

1 Overloading is defined for subprograms, enumeration literals, operators, and single entries, and also for the operations that are inherent in several basic operations such as assignment, membership tests, allocators, the literal **null**, aggregates, and string literals.

2 For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier has, whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence; overload resolution likewise determines the actual meaning of an occurrence of an operator or some basic operation.

3 At such a place all visible declarations are considered. The occurrence is only legal if there is exactly one interpretation of each constituent of the innermost complete context; a *complete context* is one of the following:[10]

4 • A declaration.

5 • A statement.

6 • A representation clause.

---

[9] See also Appendix G, AI-00192.
[10] See also Appendix G, AI-00120.

7   When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below.[11]

8   (a) Any rule that requires a name or expression to have a certain type, or to have the same type as another name or expression.[12]

9   (b) Any rule that requires the type of a name or expression to be a type of a certain class; similarly, any rule that requires a certain type to be a discrete, integer, real, universal, character, boolean, or nonlimited type.

10   (c) Any rule that requires a prefix to be appropriate for a certain type.

11   (d) Any rule that specifies a certain type as the result type of a basic operation, and any rule that specifies that this type is of a certain class.

12   (e) The rules that require the type of an aggregate or string literal to be determinable solely from the enclosing complete context (see 4.3 and 4.2). Similarly, the rules that require the type of the prefix of an attribute, the type of the expression of a case statement, or the type of the operand of a type conversion, to be determinable independently of the context (see 4.1.4, 5.4, 4.6, and 6.4.1).

13   (f) The rules given in section 6.6, for the resolution of overloaded subprogram calls; in section 4.6, for the implicit conversions of universal expressions; in section 3.6.1, for the interpretation of discrete ranges with bounds having a universal type; and in section 4.1.3, for the interpretation of an expanded name whose prefix denotes a subprogram or an accept statement.[13]

14   Subprogram names used as pragma arguments follow a different rule: the pragma can apply to several overloaded subprograms, as explained in section 6.3.2 for the pragma INLINE, in section 11.7 for the pragma SUPPRESS, and in section 13.9 for the pragma INTERFACE.

15   Similarly, the simple names given in context clauses (see 10.1.1) and in address clauses (see 13.5) follow different rules.

---

[11] See also Appendix G, AI-00157.
[12] See also Appendix G, AI-00193.
[13] See also Appendix G, AI-00287.

Notes:

16 If there is only one possible interpretation, the identifier denotes the corresponding entity. However, this does not mean that the occurrence is necessarily legal since other requirements exist which are not considered for overload resolution; for example, the fact that an expression is static, the parameter modes, whether an object is constant, conformance rules, forcing occurrences for a representation clause, order of elaboration, and so on.

17 Similarly, subtypes are not considered for overload resolution (the violation of a constraint does not make a program illegal but raises an exception during program execution).

18 Rules that require certain constructs to have the same parameter and result type profile fall under the category (a); the same holds for rules that require conformance of two constructs since conformance requires that corresponding names be given the same meaning by the visibility and overloading rules.

19 A loop parameter specification is a declaration, and hence a complete context.

20 **References:** aggregate 4.3, allocator 4.8, assignment 5.2, basic operation 3.3.3, case statement 5.4, class of type 3.3, declaration 3.1, entry 9.5, enumeration literal 3.5.1, exception 11, expression 4.4, formal part 6.1, identifier 2.3, legal 1.6, literal 4.2, loop parameter specification 5.5, membership test 4.5.2, name 4.1, null literal 3.8, operation 3.3.3, operator 4.5, overloading 6.6, pragma 2.8, representation clause 13.1, statement 5, static expression 4.9, static subtype 4.9, subprogram 6, subtype 3.3, type conversion 4.6, visibility 8.3

21 **Rule of the form (a):** address clause 13.5, assignment 5.2, choice 3.7.3 4.3.2 5.4, component association 4.3.1 4.3.2, conformance rules 9.5, default expression 3.7 3.7.1 6.1 12.1.1, delay statement 9.6, discrete range 3.6.1 5.5 9.5, discriminant constraint 3.7.2, enumeration representation clause 13.3, generic parameter association 12.3.1, index constraint 3.6.1, index expression 4.1.1 4.1.2 9.5, initial value 3.2.1, membership test 4.5.2, parameter association 6.4.1, parameter and result type profile 8.5 12.3.6, qualified expression 4.7, range constraint 3.5, renaming of an object 8.5, result expression 5.8

22 **Rules of the form (b):** abort statement 9.10, assignment 5.2, case expression 5.4, condition 5.3 5.5 5.7 9.7.1, discrete range 3.6.1 5.5 9.5, fixed point type declaration 3.5.9, floating point type declaration 3.5.7, integer type declaration 3.5.4, length clause 13.2, membership test 4.4, number declaration 3.2.2, record representation clause 13.4, selected component 4.1.3, short-circuit control form 4.4, val attribute 3.5.5

23 **Rules of the form (c):** indexed component 4.1.1, selected component 4.1.3, slice 4.1.2

24 **Rules of the form (d):** aggregate 4.3, allocator 4.8, membership test 4.4, null literal 4.2, numeric literal 2.4, short-circuit control form 4.4, string literal 4.2

9
Tasks

1   The execution of a program that does not contain a task is defined in terms of a sequential execution of its actions, according to the rules described in other chapters of this manual. These actions can be considered to be executed by a single *logical processor.*

2   Tasks are entities whose executions proceed *in parallel* in the following sense. Each task can be considered to be executed by a logical processor of its own. Different tasks (different logical processors) proceed independently, except at points where they synchronize.

3   Some tasks have *entries.* An entry of a task can be *called* by other tasks. A task *accepts* a call of one of its entries by executing an accept statement for the entry. Synchronization is achieved by *rendezvous* between a task issuing an entry call and a task accepting the call. Some entries have parameters; entry calls and accept statements for such entries are the principal means of communicating values between tasks.

4   The properties of each task are defined by a corresponding *task unit* which consists of a *task specification* and a *task body.* Task units are one of the four forms of program unit of which programs can be composed. The other forms are subprograms, packages and generic units. The properties of task units, tasks, and entries, and the statements that affect the interaction between tasks (that is, entry call statements, accept statements, delay statements, select statements, and abort statements) are described in this chapter.

Note:

5   Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single *physical processor.* On the other hand, whenever an implementation can detect that the same effect can be guaranteed if parts of the actions of a given task are executed by different physical processors acting in parallel, it may choose to execute them in this way; in such a case, several physical processors implement a single logical processor.

6 **References** : abort statement 9.10, accept statement 9.5, delay statement 9.6, entry 9.5, entry call statement 9.5, generic unit 12, package 7, parameter in an entry call 9.5, program unit 6, rendezvous 9.5, select statement 9.7, subprogram 6, task body 9.1, task specification 9.1

9.1 Task Specifications and Task Bodies

1 A task unit consists of a task specification and a task body. A task specification that starts with the reserved words **task type** declares a task type. The value of an object of a task type designates a task having the entries, if any, that are declared in the task specification; these entries are also called entries of this object. The execution of the task is defined by the corresponding task body.

2 A task specification without the reserved word **type** defines a *single task.* A task declaration with this form of specification is equivalent to the declaration of an anonymous task type immediately followed by the declaration of an object of the task type, and the task unit identifier names the object. In the remainder of this chapter, explanations are given in terms of task type declarations; the corresponding explanations for single task declarations follow from the stated equivalence.

```
3     task_declaration ::= task_specification;

      task_specification ::=
         task [type] identifier [is
            {entry_declaration}
            {representation_clause}
         end [task_simple_name]]

      task_body ::=
          task body task_simple_name is
            [declarative_part]
          begin
              sequence_of_statements
         [exception
              exception_handler
            {exception_handler}]
          end [task_simple_name];
```

4 The simple name at the start of a task body must repeat the task unit identifier. Similarly if a simple name appears at the end of the task specification or body, it must repeat the task unit identifier. Within a task body, the name of the corresponding task unit can also be used to refer to the task object that designates the task currently executing the body; furthermore, the use of this name as a type mark is not allowed within the task unit itself.

5   For the elaboration of a task specification, entry declarations and
     representation clauses, if any, are elaborated in the order given. Such
     representation clauses only apply to the entries declared in the task
     specification (see 13.5).

6   The elaboration of a task body has no other effect than to establish that the
     body can from then on be used for the execution of tasks designated by objects
     of the corresponding task type.

7   The execution of a task body is invoked by the activation of a task object of
     the corresponding type (see 9.3). The optional exception handlers at the end of
     a task body handle exceptions raised during the execution of the sequence of
     statements of the task body (see 11.4).

8   **Examples of specifications of task types:**

```
task type RESOURCE is
    entry SEIZE;
    entry RELEASE;
end RESOURCE;

task type KEYBOARD_DRIVER is
    entry READ (C : out CHARACTER);
    entry WRITE(C : in  CHARACTER);
end KEYBOARD_DRIVER;
```

9   **Examples of specifications of single tasks:**

```
task PRODUCER_CONSUMER is
    entry READ (V : out ITEM);
    entry WRITE(E : in  ITEM);
end;

task CONTROLLER is
    entry REQUEST(LEVEL)(D : ITEM);  --  a family of entries
end CONTROLLER;
task USER;  --  has no entries
```

10   **Example of task specification and corresponding body:**

```
task PROTECTED_ARRAY is
    --  INDEX and ITEM are global types
    entry READ (N : in INDEX; V : out ITEM);
    entry WRITE(N : in INDEX; E : in  ITEM);
end;
```

```
task body PROTECTED_ARRAY is
   TABLE : array(INDEX) of ITEM := (INDEX => NULL_ITEM);
begin
   loop
      select
         accept READ (N : in INDEX; V : out ITEM) do
            V := TABLE(N);
         end READ;
      or
         accept WRITE(N : in INDEX; E : in  ITEM) do
            TABLE(N) := E;
         end WRITE;
      end select;
   end loop;
end PROTECTED_ARRAY;
```

Note:

11   A task specification specifies the interface of tasks of the task type with other
      tasks of the same or of different types, and also with the main program.

12   **References** : declaration 3.1, declarative part 3.9, elaboration 3.9, entry 9.5, entry
      declaration 9.5, exception handler 11.2, identifier 2.3, main program 10.1, object 3.2,
      object declaration 3.2.1, representation clause 13.1, reserved word 2.9, sequence of
      statements 5.1, simple name 4.1, type 3.3, type declaration 3.3.1

9.2 Task Types and Task Objects

1   A task type is a limited type (see 7.4.4). Hence neither assignment nor the
      predefined comparison for equality and inequality are defined for objects of
      task types; moreover, the mode **out** is not allowed for a formal parameter
      whose type is a task type.

2   A task object is an object whose type is a task type. The value of a task object
      designates a task that has the entries of the corresponding task type, and
      whose execution is specified by the corresponding task body. If a task object is
      the object, or a subcomponent of the object, declared by an object declaration,
      then the value of the task object is defined by the elaboration of the object
      declaration. If a task object is the object, or a subcomponent of the object,
      created by the evaluation of an allocator, then the value of the task object is
      defined by the evaluation of the allocator. For all parameter modes, if an actual
      parameter designates a task, the associated formal parameter designates the
      same task; the same holds for a subcomponent of an actual parameter and the
      corresponding subcomponent of the associated formal parameter; finally, the
      same holds for generic parameters.

3 **Examples:**
```
CONTROL  : RESOURCE;
TELETYPE : KEYBOARD_DRIVER;
POOL     : array(1 .. 10) of KEYBOARD_DRIVER;
-- see also examples of declarations of single tasks in 9.1
```

4 **Example of access type designating task objects:**
```
type KEYBOARD is access KEYBOARD_DRIVER;
```

```
TERMINAL : KEYBOARD := new KEYBOARD_DRIVER;
```

Notes:

5 Since a task type is a limited type, it can appear as the definition of a limited private type in a private part, and as a generic actual parameter associated with a formal parameter whose type is a limited type. On the other hand, the type of a generic formal parameter of mode **in** must not be a limited type and hence cannot be a task type.

6 Task objects behave as constants (a task object always designates the same task) since their values are implicitly defined either at declaration or allocation, or by a parameter association, and since no assignment is available. However the reserved word **constant** is not allowed in the declaration of a task object since this would require an explicit initialization. A task object that is a formal parameter of mode **in** is a constant (as is any formal parameter of this mode).

7 If an application needs to store and exchange task identities, it can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes (see above example). Assignment is available for such an access type as for any access type.

8 Subtype declarations are allowed for task types as for other types, but there are no constraints applicable to task types.

9 **References** : access type 3.8, actual parameter 6.4.1, allocator 4.8, assignment 5.2, component declaration 3.7, composite type 3.3, constant 3.2.1, constant declaration 3.2.1, constraint 3.3, designate 3.8 9.1, elaboration 3.9, entry 9.5, equality operator 4.5.2, formal parameter 6.2, formal parameter mode 6.2, generic actual parameter 12.3, generic association 12.3, generic formal parameter 12.1, generic formal parameter mode 12.1.1, generic unit 12, inequality operator 4.5.2, initialization 3.2.1, limited type 7.4.4, object 3.2, object declaration 3.2.1, parameter association 6.4, private part 7.2, private type 7.4, reserved word 2.9, subcomponent 3.3, subprogram 6, subtype declaration 3.3.2, task body 9.1, type 3.3

9.3 Task Execution—Task Activation

1   A task body defines the execution of any task that is designated by a task object of the corresponding task type. The initial part of this execution is called the *activation* of the task object, and also that of the designated task; it consists of the elaboration of the declarative part, if any, of the task body. The execution of different tasks, in particular their activation, proceeds in parallel.

2   If an object declaration that declares a task object occurs immediately within a declarative part, then the activation of the task object starts after the elaboration of the declarative part (that is, after passing the reserved word **begin** following the declarative part); similarly if such a declaration occurs immediately within a package specification, the activation starts after the elaboration of the declarative part of the package body. The same holds for the activation of a task object that is a subcomponent of an object declared immediately within a declarative part or package specification. The first statement following the declarative part is executed only after conclusion of the activation of these task objects.

3   Should an exception be raised by the activation of one of these tasks, that task becomes a completed task (see 9.4); other tasks are not directly affected. Should one of these tasks thus become completed during its activation, the exception TASKING_ERROR is raised upon conclusion of the activation of all of these tasks (whether successfully or not); the exception is raised at a place that is immediately before the first statement following the declarative part (immediately after the reserved word **begin**). Should several of these tasks thus become completed during their activation, the exception TASKING_ ERROR is raised only once.[1]

4   Should an exception be raised by the elaboration of a declarative part or package specification, then any task that is created (directly or indirectly) by this elaboration and that is not yet activated becomes terminated and is therefore never activated (see section 9.4 for the definition of a terminated task).[2]

5   For the above rules, in any package body without statements, a null statement is assumed. For any package without a package body, an implicit package body containing a single null statement is assumed. If a package without a package body is declared immediately within some program unit or block statement, the implicit package body occurs at the end of the declarative part of the program unit or block statement; if there are several such packages, the order of the implicit package bodies is undefined.[3]

---

[1]   See also Appendix G, AI-00268.
[2]   See also Appendix G, AI-00198.
[3]   See also Appendix G, AI-00237.

6   A task object that is the object, or a subcomponent of the object, created by
     the evaluation of an allocator is activated by this evaluation. The activation
     starts after any initialization for the object created by the allocator; if several
     subcomponents are task objects, they are activated in parallel. The access
     value designating such an object is returned by the allocator only after the
     conclusion of these activations.

7   Should an exception be raised by the activation of one of these tasks, that
     task becomes a completed task; other tasks are not directly affected. Should
     one of these tasks thus become completed during its activation, the exception
     TASKING_ERROR is raised upon conclusion of the activation of all of these
     tasks (whether successfully or not); the exception is raised at the place where
     the allocator is evaluated. Should several of these tasks thus become completed
     during their activation, the exception TASKING_ERROR is raised only once.

8   Should an exception be raised by the initialization of the object created by
     an allocator (hence before the start of any activation), any task designated
     by a subcomponent of this object becomes terminated and is therefore never
     activated.[4]

9   **Example:**

```
procedure P is
    A, B : RESOURCE;  --  elaborate the task objects A, B
    C    : RESOURCE;  --  elaborate the task object C
begin
    --  the tasks A, B, C are activated in parallel
    --  before the first statement
    ...
end;
```

     Notes:

10   An entry of a task can be called before the task has been activated. If several
     tasks are activated in parallel, the execution of any of these tasks need
     not await the end of the activation of the other tasks. A task may become
     completed during its activation either because of an exception or because it is
     aborted (see 9.10).

11   **References** : allocator 4.8, completed task 9.4, declarative part 3.9, elaboration
     3.9, entry 9.5, exception 11, handling an exception 11.4, package body 7.1, parallel
     execution 9, statement 5, subcomponent 3.3, task body 9.1, task object 9.2, task
     termination 9.4, task type 9.1, tasking_error exception 11.1

---

[4] See also Appendix G, AI-00198.

9.4 Task Dependence—Termination of Tasks

1 Each task *depends* on at least one master.[5] A *master* is a construct that is either a task, a currently executing block statement or subprogram, or a library package (a package declared within another program unit is not a master). The dependence on a master is a direct dependence in the following two cases:

2   (a)   The task designated by a task object that is the object, or a subcomponent of the object, created by the evaluation of an allocator depends on the master that elaborates the corresponding access type definition.

3   (b)   The task designated by any other task object depends on the master whose execution creates the task object.

4 Furthermore, if a task depends on a given master that is a block statement executed by another master, then the task depends also on this other master, in an indirect manner; the same holds if the given master is a subprogram called by another master, and if the given master is a task that depends (directly or indirectly) on another master. Dependences exist for objects of a private type whose full declaration is in terms of a task type.

5 A task is said to have *completed* its execution when it has finished the execution of the sequence of statements that appears after the reserved word **begin** in the corresponding body. Similarly a block or a subprogram is said to have completed its execution when it has finished the execution of the corresponding sequence of statements. For a block statement, the execution is also said to be completed when it reaches an exit, return, or goto statement transferring control out of the block. For a procedure, the execution is also said to be completed when a corresponding return statement is reached. For a function, the execution is also said to be completed after the evaluation of the result expression of a return statement. Finally the execution of a task, block statement, or subprogram is completed if an exception is raised by the execution of its sequence of statements and there is no corresponding handler, or, if there is one, when it has finished the execution of the corresponding handler.[6]

6 If a task has no dependent task, its *termination* takes place when it has completed its execution. After its termination, a task is said to be *terminated.* If a task has dependent tasks, its termination takes place when the execution of the task is completed and all dependent tasks are terminated. A block

---

[5] See also Appendix G, AI-00167.
[6] See also Appendix G, AI-00173.

statement or subprogram body whose execution is completed is not left until all of its dependent tasks are terminated.[7]

7   Termination of a task otherwise takes place if and only if its execution has reached an open terminate alternative in a select statement (see 9.7.1), and the following conditions are satisfied:

8   • The task depends on some master whose execution is completed (hence not a library package).

9   • Each task that depends on the master considered is either already terminated or similarly waiting on an open terminate alternative of a select statement.

10 When both conditions are satisfied, the task considered becomes terminated, together with all tasks that depend on the master considered.

11 **Example:**

```
declare
   type GLOBAL is access RESOURCE;          -- see 9.1
   A, B : RESOURCE;
   G    : GLOBAL;
begin
   --  activation of A and B
   declare
      type LOCAL is access RESOURCE;
      X : GLOBAL := new RESOURCE;  --  activation of X.all
      L : LOCAL  := new RESOURCE;  --  activation of L.all
      C : RESOURCE;
   begin
      --  activation of C
      G := X;  --  both G and X designate the same task object
      ...
   end;  --  await termination of C and L.all (but not X.all)
   ...
end;  --  await termination of A, B, and G.all
```

Notes:

12 The rules given for termination imply that all tasks that depend (directly or indirectly) on a given master and that are not already terminated, can be terminated (collectively) if and only if each of them is waiting on an open terminate alternative of a select statement and the execution of the given master is completed.

---

[7]   See also Appendix G, AI-00441.

13   The usual rules apply to the main program. Consequently, termination of the main program awaits termination of any dependent task even if the corresponding task type is declared in a library package. On the other hand, termination of the main program does not await termination of tasks that depend on library packages; the language does not define whether such tasks are required to terminate.

In DEC Ada, the environment task that calls the main program (see 10.1) is the master of tasks that depend on library packages. In accordance with the rules of this section, the environment task awaits termination of such tasks. In particular, the rules concerning terminate alternatives in select statements apply.

In mixed-language programs, termination of Ada tasks takes place according to the usual rules. For example, termination of a non-Ada main program awaits termination of any Ada subprograms that involve tasks.

14   For an access type derived from another access type, the corresponding access type definition is that of the parent type; the dependence is on the master that elaborates the ultimate parent access type definition.

15   A renaming declaration defines a new name for an existing entity and hence creates no further dependence.

16   **References** : access type 3.8, allocator 4.8, block statement 5.6, declaration 3.1, designate 3.8 9.1, exception 11, exception handler 11.2, exit statement 5.7, function 6.5, goto statement 5.9, library unit 10.1, main program 10.1, object 3.2, open alternative 9.7.1, package 7, program unit 6, renaming declaration 8.5, return statement 5.8, selective wait 9.7.1, sequence of statements 5.1, statement 5, subcomponent 3.3, subprogram body 6.3, subprogram call 6.4, task body 9.1, task object 9.2, terminate alternative 9.7.1

9.5 Entries, Entry Calls, and Accept Statements

1   Entry calls and accept statements are the primary means of synchronization of tasks, and of communicating values between tasks. An entry declaration is similar to a subprogram declaration and is only allowed in a task specification. The actions to be performed when an entry is called are specified by corresponding accept statements.

```
2       entry_declaration ::=
            entry identifier [(discrete_range)] [formal_part];

        entry_call_statement ::= entry_name [actual_parameter_part];
```

```
accept_statement ::=
    accept entry_simple_name [(entry_index)] [formal_part] [do
        sequence_of_statements
    end [entry_simple_name]];

entry_index ::= expression
```

3   An entry declaration that includes a discrete range (see 3.6.1) declares a *family* of distinct entries having the same formal part (if any); that is, one such entry for each value of the discrete range. The term *single entry* is used in the definition of any rule that applies to any entry other than one of a family. The task designated by an object of a task type has (or owns) the entries declared in the specification of the task type.

4   Within the body of a task, each of its single entries or entry families can be named by the corresponding simple name. The name of an entry of a family takes the form of an indexed component, the family simple name being followed by the index in parentheses; the type of this index must be the same as that of the discrete range in the corresponding entry family declaration. Outside the body of a task an entry name has the form of a selected component, whose prefix denotes the task object, and whose selector is the simple name of one of its single entries or entry families.

5   A single entry overloads a subprogram, an enumeration literal, or another single entry if they have the same identifier. Overloading is not defined for entry families. A single entry or an entry of an entry family can be renamed as a procedure as explained in section 8.5.[8]

6   The parameter modes defined for parameters of the formal part of an entry declaration are the same as for a subprogram declaration and have the same meaning (see 6.2). The syntax of an entry call statement is similar to that of a procedure call statement, and the rules for parameter associations are the same as for subprogram calls (see 6.4.1 and 6.4.2).

7   An accept statement specifies the actions to be performed at a call of a named entry (it can be an entry of a family). The formal part of an accept statement must conform to the formal part given in the declaration of the single entry or entry family named by the accept statement (see section 6.3.1 for the conformance rules). If a simple name appears at the end of an accept statement, it must repeat that given at the start.

8   An accept statement for an entry of a given task is only allowed within the corresponding task body; excluding within the body of any program unit that is, itself, inner to the task body; and excluding within another accept statement for either the same single entry or an entry of the same family.

---

[8]   See also Appendix G, AI-00287.

(One consequence of this rule is that a task can execute accept statements only for its own entries.) A task body can contain more than one accept statement for the same entry.

9   For the elaboration of an entry declaration, the discrete range, if any, is evaluated and the formal part, if any, is then elaborated as for a subprogram declaration.

10 Execution of an accept statement starts with the evaluation of the entry index (in the case of an entry of a family). Execution of an entry call statement starts with the evaluation of the entry name; this is followed by any evaluations required for actual parameters in the same manner as for a subprogram call (see 6.4). Further execution of an accept statement and of a corresponding entry call statement are synchronized.

11 If a given entry is called by only one task, there are two possibilities:

12 •   If the calling task issues an entry call statement before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is *suspended.*

13 •   If a task reaches an accept statement prior to any call of that entry, the execution of the task is suspended until such a call is received.

14 When an entry has been called and a corresponding accept statement has been reached, the sequence of statements, if any, of the accept statement is executed by the called task (while the calling task remains suspended). This interaction is called a *rendezvous.* Thereafter, the calling task and the task owning the entry continue their execution in parallel.

15 If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued; there is one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in the order of arrival.

16 An attempt to call an entry of a task that has completed its execution raises the exception TASKING_ERROR at the point of the call, in the calling task; similarly, this exception is raised at the point of the call if the called task completes its execution before accepting the call (see also 9.10 for the case when the called task becomes abnormal). The exception CONSTRAINT_ERROR is raised if the index of an entry of a family is not within the specified discrete range.

17 **Examples of entry declarations:**
```
entry READ(V : out ITEM);
entry SEIZE;
entry REQUEST(LEVEL)(D : ITEM);  --  a family of entries
```

18 **Examples of entry calls:**

```
CONTROL.RELEASE;                          -- see 9.2 and 9.1
PRODUCER_CONSUMER.WRITE(E);               -- see 9.1
POOL(5).READ(NEXT_CHAR);                  -- see 9.2 and 9.1
CONTROLLER.REQUEST(LOW)(SOME_ITEM);       -- see 9.1
```

19 **Examples of accept statements:**

```
accept SEIZE;

accept READ(V : out ITEM) do
   V := LOCAL_ITEM;
end READ;

accept REQUEST(LOW)(D : ITEM) do
   ...
end REQUEST;
```

Notes:

20 The formal part given in an accept statement is not elaborated; it is only used to identify the corresponding entry.

21 An accept statement can call subprograms that issue entry calls. An accept statement need not have a sequence of statements even if the corresponding entry has parameters. Equally, it can have a sequence of statements even if the corresponding entry has no parameters. The sequence of statements of an accept statement can include return statements. A task can call its own entries but it will, of course, deadlock. The language permits conditional and timed entry calls (see 9.7.2 and 9.7.3). The language rules ensure that a task can only be in one entry queue at a given time.

22 If the bounds of the discrete range of an entry family are integer literals, the index (in an entry name or accept statement) must be of the predefined type INTEGER (see 3.6.1).

23 **References** : abnormal task 9.10, actual parameter part 6.4, completed task 9.4, conditional entry call 9.7.2, conformance rules 6.3.1, constraint_error exception 11.1, designate 9.1, discrete range 3.6.1, elaboration 3.1 3.9, enumeration literal 3.5.1, evaluation 4.5, expression 4.4, formal part 6.1, identifier 2.3, indexed component 4.1.1, integer type 3.5.4, name 4.1, object 3.2, overloading 6.6 8.7, parallel execution 9, prefix 4.1, procedure 6, procedure call 6.4, renaming declaration 8.5, return statement 5.8, scope 8.2, selected component 4.1.3, selector 4.1.3, sequence of statements 5.1, simple expression 4.4, simple name 4.1, subprogram 6, subprogram body 6.3, subprogram declaration 6.1, task 9, task body 9.1, task specification 9.1, tasking_error exception 11.1, timed entry call 9.7.3

9.6 Delay Statements, Duration, and Time

1   The execution of a delay statement evaluates the simple expression, and suspends further execution of the task that executes the delay statement, for at least the duration specified by the resulting value.[9]

2   `delay_statement ::= `**`delay`**` simple_expression;`

3   The simple expression must be of the predefined fixed point type DURATION; its value is expressed in seconds; a delay statement with a negative value is equivalent to a delay statement with a zero value.

4   Any implementation of the type DURATION must allow representation of durations (both positive and negative) up to at least 86400 seconds (one day); the smallest representable duration, DURATION′SMALL must not be greater than twenty milliseconds (whenever possible, a value not greater than fifty microseconds should be chosen). Note that DURATION′SMALL need not correspond to the basic clock cycle, the named number SYSTEM.TICK (see 13.7).[10]

In DEC Ada, DURATION′SMALL is 2^{-14} seconds or approximately 61 microseconds. The value does not correspond to the value of the named number SYSTEM.TICK.

On OpenVMS VAX systems, the value of SYSTEM.TICK is 10.0^{-2} seconds or 10 milliseconds. (SYSTEM.TICK represents the smallest unit of time used by the OpenVMS VAX operating system in its time-related system services.)

On Alpha systems, the value of SYSTEM.TICK is 10.0^{-3} seconds or 1 millisecond. (On OpenVMS Alpha systems, SYSTEM.TICK represents the smallest unit of time used by the operating system in its time-related system services. On Digital UNIX Alpha systems, SYSTEM.TICK represents the resolution of the Digital UNIX system clock.)

5   The definition of the type TIME is provided in the predefined library package CALENDAR. The function CLOCK returns the current value of TIME at the time it is called. The functions YEAR, MONTH, DAY and SECONDS return the corresponding values for a given value of the type TIME; the procedure SPLIT returns all four corresponding values. Conversely, the function TIME_OF combines a year number, a month number, a day number, and a duration, into a value of type TIME. The operators "+" and "−" for addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning.[11]

---

[9] See also Appendix G, AI-00201 and AI-00464.
[10] See also Appendix G, AI-00201.
[11] See also Appendix G, AI-00195 and AI-00201.

On OpenVMS systems, the function CLOCK returns the current time, and the type TIME is implemented as OpenVMS binary time.

On Digital UNIX systems, the function CLOCK returns Greenwich time, and the type TIME is implemented as a record type with two components. The first component expresses the number of days since 1 January 1901. The second component expresses the number of seconds that have passed during the current day.

6   The exception TIME_ERROR is raised by the function TIME_OF if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "−" if, for the given operands, these operators cannot return a date whose year number is in the range of the corresponding subtype, or if the operator "−" cannot return a result that is in the range of the type DURATION.[12]

7
```ada
package CALENDAR is
   type TIME is private;

   subtype YEAR_NUMBER  is INTEGER  range 1901 .. 2099;
   subtype MONTH_NUMBER is INTEGER  range 1 .. 12;
   subtype DAY_NUMBER   is INTEGER  range 1 .. 31;
   subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;

   function CLOCK return TIME;

   function YEAR   (DATE : TIME) return YEAR_NUMBER;
   function MONTH  (DATE : TIME) return MONTH_NUMBER;
   function DAY    (DATE : TIME) return DAY_NUMBER;
   function SECONDS(DATE : TIME) return DAY_DURATION;

   procedure SPLIT (DATE    : in  TIME;
                    YEAR    : out YEAR_NUMBER;
                    MONTH   : out MONTH_NUMBER;
                    DAY     : out DAY_NUMBER;
                    SECONDS : out DAY_DURATION);

   function TIME_OF(YEAR    : YEAR_NUMBER;
                    MONTH   : MONTH_NUMBER;
                    DAY     : DAY_NUMBER;
                    SECONDS : DAY_DURATION := 0.0) return TIME;

   function "+" (LEFT  : TIME;
                 RIGHT : DURATION) return TIME;
   function "+" (LEFT  : DURATION;
                 RIGHT : TIME)     return TIME;
   function "-" (LEFT  : TIME;
                 RIGHT : DURATION) return TIME;
   function "-" (LEFT  : TIME;
                 RIGHT : TIME)     return DURATION;
```

---

[12] See also Appendix G, AI-00196.

```
   function "<"  (LEFT, RIGHT : TIME) return BOOLEAN;
   function "<=" (LEFT, RIGHT : TIME) return BOOLEAN;
   function ">"  (LEFT, RIGHT : TIME) return BOOLEAN;
   function ">=" (LEFT, RIGHT : TIME) return BOOLEAN;

   TIME_ERROR : exception;
   --  can be raised by TIME_OF, "+", and "-"
private
   -- implementation-dependent
end;13
```

8   **Examples:**

```
delay 3.0;  --  delay 3.0 seconds

declare
   use CALENDAR;
   --  INTERVAL is a global constant of type DURATION
   NEXT_TIME : TIME := CLOCK + INTERVAL;
begin
   loop
      delay NEXT_TIME - CLOCK;
      --  some actions
      NEXT_TIME := NEXT_TIME + INTERVAL;
   end loop;
end;
```

Notes:

9   The second example causes the loop to be repeated every INTERVAL
seconds on average. This interval between two successive iterations is only
approximate. However, there will be no cumulative drift as long as the
duration of each iteration is (sufficiently) less than INTERVAL.

10   **References** : adding operator 4.5, duration C, fixed point type 3.5.9, function call
6.4, library unit 10.1, operator 4.5, package 7, private type 7.4, relational operator 4.5,
simple expression 4.4, statement 5, task 9, type 3.3

named number 3.2, system predefined package 13.7 13.7.1

---

13 See also Appendix G, AI-00355.

9.7 Select Statements

1   There are three forms of select statements. One form provides a selective wait
    for one or more alternatives. The other two provide conditional and timed
    entry calls.

2   ```
    select_statement ::= selective_wait
        | conditional_entry_call | timed_entry_call
    ```

3   **References** : selective wait 9.7.1, conditional entry call 9.7.2, timed entry call 9.7.3

## 9.7.1 Selective Waits

1   This form of the select statement allows a combination of waiting for, and
    selecting from, one or more alternatives. The selection can depend on
    conditions associated with each alternative of the selective wait.

2   ```
    selective_wait ::=
         select
           select_alternative
        {or
           select_alternative}
        [else
           sequence_of_statements]
         end select;

    select_alternative ::=
       [when condition =>]
          selective_wait_alternative

    selective_wait_alternative ::= accept_alternative
       | delay_alternative | terminate_alternative

    accept_alternative ::= accept_statement [sequence_of_statements]

    delay_alternative  ::= delay_statement  [sequence_of_statements]

    terminate_alternative ::= terminate;
    ```

3   A selective wait must contain at least one accept alternative. In addition a
    selective wait can contain either a terminate alternative (only one), or one or
    more delay alternatives, or an else part; these three possibilities are mutually
    exclusive.

4   A select alternative is said to be *open* if it does not start with **when** and a
    condition, or if the condition is TRUE. It is said to be *closed* otherwise.

5   For the execution of a selective wait, any conditions specified after **when** are
    evaluated in some order that is not defined by the language; open alternatives
    are thus determined. For an open delay alternative, the delay expression
    is also evaluated. Similarly, for an open accept alternative for an entry of a
    family, the entry index is also evaluated. Selection and execution of one open

alternative, or of the else part, then completes the execution of the selective wait; the rules for this selection are described below.[14]

6   Open accept alternatives are first considered. Selection of one such alternative takes place immediately if a corresponding rendezvous is possible, that is, if there is a corresponding entry call issued by another task and waiting to be accepted. If several alternatives can thus be selected, one of them is selected arbitrarily (that is, the language does not define which one). When such an alternative is selected, the corresponding accept statement and possible subsequent statements are executed. If no rendezvous is immediately possible and there is no else part, the task waits until an open selective wait alternative can be selected.

7   Selection of the other forms of alternative or of an else part is performed as follows:

8   • An open delay alternative will be selected if no accept alternative can be selected before the specified delay has elapsed (immediately, for a negative or zero delay in the absence of queued entry calls); any subsequent statements of the alternative are then executed. If several delay alternatives can thus be selected (that is, if they have the same delay), one of them is selected arbitrarily.

9   • The else part is selected and its statements are executed if no accept alternative can be immediately selected, in particular, if all alternatives are closed.

10   • An open terminate alternative is selected if the conditions stated in section 9.4 are satisfied. It is a consequence of other rules that a terminate alternative cannot be selected while there is a queued entry call for any entry of the task.

11   The exception PROGRAM_ERROR is raised if all alternatives are closed and there is no else part.

12   **Examples of a select statement:**
```
select
    accept DRIVER_AWAKE_SIGNAL;
or
    delay 30.0*SECONDS;
    STOP_THE_TRAIN;
end select;
```

---

[14] See also Appendix G, AI-00030.

13   **Example of a task body with a select statement:**

```
task body RESOURCE is
   BUSY : BOOLEAN := FALSE;
begin
   loop
      select
         when not BUSY =>
            accept SEIZE do
               BUSY := TRUE;
            end;
      or
         accept RELEASE do
            BUSY := FALSE;
         end;
      or
         terminate;
      end select;
   end loop;
end RESOURCE;
```

   Notes:

14   A selective wait is allowed to have several open delay alternatives. A selective
     wait is allowed to have several open accept alternatives for the same entry.

15   **References** : accept statement 9.5, condition 5.3, declaration 3.1, delay expression
     9.6, delay statement 9.6, duration 9.6, entry 9.5, entry call 9.5, entry index 9.5,
     program_error exception 11.1, queued entry call 9.5, rendezvous 9.5, select statement
     9.7, sequence of statements 5.1, task 9

9.7.2 Conditional Entry Calls

1   A conditional entry call issues an entry call that is then canceled if a
    rendezvous is not immediately possible.[15]

2   ```
    conditional_entry_call ::=
        select
            entry_call_statement
          [sequence_of_statements]
        else
            sequence_of_statements
        end select;
    ```

3   For the execution of a conditional entry call, the entry name is first evaluated.
    This is followed by any evaluations required for actual parameters as in the
    case of a subprogram call (see 6.4).

---

[15]  See also Appendix G, AI-00276 and AI-00444.

4   The entry call is canceled if the execution of the called task has not reached a point where it is ready to accept the call (that is, either an accept statement for the corresponding entry, or a select statement with an open accept alternative for the entry), or if there are prior queued entry calls for this entry. If the called task has reached a select statement, the entry call is canceled if an accept alternative for this entry is not selected.

5   If the entry call is canceled, the statements of the else part are executed. Otherwise, the rendezvous takes place; and the optional sequence of statements after the entry call is then executed.

6   The execution of a conditional entry call raises the exception TASKING_ ERROR if the called task has already completed its execution (see also 9.10 for the case when the called task becomes abnormal).

7   **Example:**

```
procedure SPIN(R : RESOURCE) is
begin
   loop
      select
         R.SEIZE;
         return;
      else
         null;  --  busy waiting
      end select;
   end loop;
end;
```

8   **References** : abnormal task 9.10, accept statement 9.5, actual parameter part 6.4, completed task 9.4, entry call statement 9.5, entry family 9.5, entry index 9.5, evaluation 4.5, expression 4.4, open alternative 9.7.1, queued entry call 9.5, rendezvous 9.5, select statement 9.7, sequence of statements 5.1, task 9, tasking_error exception 11.1

9.7.3 Timed Entry Calls

1   A timed entry call issues an entry call that is canceled if a rendezvous is not started within a given delay.

```
2       timed_entry_call ::=
           select
               entry_call_statement
             [sequence_of_statements]
           or
               delay_alternative
           end select;
```

3   For the execution of a timed entry call, the entry name is first evaluated. This is followed by any evaluations required for actual parameters as in the case of a subprogram call (see 6.4). The expression stating the delay is then evaluated, and the entry call is finally issued.

4   If a rendezvous can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed and the optional sequence of statements after the entry call is then executed. Otherwise, the entry call is canceled when the specified duration has expired, and the optional sequence of statements of the delay alternative is executed.[16]

5   The execution of a timed entry call raises the exception TASKING_ERROR if the called task completes its execution before accepting the call (see also 9.10 for the case when the called task becomes abnormal).

6   **Example:**
```
select
    CONTROLLER.REQUEST(MEDIUM)(SOME_ITEM);
or
    delay 45.0;
    --  controller too busy, try something else
end select;
```

7   **References** : abnormal task 9.10, accept statement 9.5, actual parameter part 6.4, completed task 9.4, conditional entry call 9.7.2, delay expression 9.6, delay statement 9.6, duration 9.6, entry call statement 9.5, entry family 9.5, entry index 9.5, evaluation 4.5, expression 4.4, rendezvous 9.5, sequence of statements 5.1, task 9, tasking_error exception 11.1

9.8 Priorities

1   Each task may (but need not) have a priority, which is a value of the subtype PRIORITY (of the type INTEGER) declared in the predefined library package SYSTEM (see 13.7).[17] A lower value indicates a lower degree of urgency; the range of priorities is implementation-defined. A priority is associated with a task if a pragma[18]

```
pragma PRIORITY (static_expression);
```

---

[16] See also Appendix G, AI-00276.
[17] See also Appendix G, AI-00197.
[18] See also Appendix G, AI-00031.

2   appears in the corresponding task specification; the priority is given by the value of the expression. A priority is associated with the main program if such a pragma appears in its outermost declarative part. At most one such pragma can appear within a given task specification or for a subprogram that is a library unit, and these are the only allowed places for this pragma. A pragma PRIORITY has no effect if it occurs in a subprogram other than the main program.

DEC Ada specifies the subtype PRIORITY to be of the type INTEGER with a range of 0 .. 15.

On Digital UNIX systems, DEC Ada allows task priorities to be changed dynamically at run time to values of the subtype PRIORITY, as well as to the values of Digital UNIX real-time and system priorities.

See the relevant DEC Ada run-time reference manual for information on using the pragma PRIORITY to control DEC Ada task scheduling. See the *DEC Ada Run-Time Reference Manual for DEC OSF/1 Systems* or the specification of the DEC Ada package SET_TASK_PRIORITY for more information on dynamically changing task priorities on Digital UNIX systems.

3   The specification of a priority is an indication given to assist the implementation in the allocation of processing resources to parallel tasks when there are more tasks eligible for execution than can be supported simultaneously by the available processing resources. The effect of priorities on scheduling is defined by the following rule:

4   If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processors and the same other processing resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.[19]

5   For tasks of the same priority, the scheduling order is not defined by the language. For tasks without explicit priority, the scheduling rules are not defined, except when such tasks are engaged in a rendezvous. If the priorities of both tasks engaged in a rendezvous are defined, the rendezvous is executed with the higher of the two priorities. If only one of the two priorities is defined, the rendezvous is executed with at least that priority. If neither is defined, the priority of the rendezvous is undefined.

For information on DEC Ada task scheduling, see the DEC Ada run-time reference manuals.

---

[19] See also Appendix G, AI-00032 and AI-00288.

Notes:

6 The priority of a task is static and therefore fixed. However, the priority during a rendezvous is not necessarily static since it also depends on the priority of the task calling the entry. Priorities should be used only to indicate relative degrees of urgency; they should not be used for task synchronization.

7 **References** : declarative part 3.9, entry call statement 9.5, integer type 3.5.4, main program 10.1, package system 13.7, pragma 2.8, rendezvous 9.5, static expression 4.9, subtype 3.3, task 9, task specification 9.1

9.8a Pragma Time_Slice (OpenVMS and Digital UNIX Systems Only)

On OpenVMS and Digital UNIX systems, two scheduling strategies are available for DEC Ada tasks:

- Tasks of the same priority are executed in first-in first-out order. This strategy is the default on OpenVMS VAX systems.

- Tasks of equal priority take turns at the processor (round-robin task scheduling). This strategy is the default on Alpha systems.

To allow round-robin scheduling to be enabled or disabled, DEC Ada provides the pragma TIME_SLICE. The form of this pragma is as follows:

```
pragma TIME_SLICE (static_expression);
```

The static expression gives the value of a time slice in seconds. It must be of the predefined fixed point type DURATION. A positive value enables round-robin scheduling. A negative or zero value disables round-robin scheduling.

On OpenVMS VAX systems, a positive time-slice value also determines the maximum number of seconds each task executes when it takes its turn at the processor.

This pragma is only allowed in the outermost declarative part of a subprogram that is a library unit. At most one such pragma is allowed in a subprogram. If it occurs in a subprogram other than the main program, this pragma has no effect.

The following rules define the effect of enabling round-robin scheduling with the pragma TIME_SLICE:

- The value applies to the scheduling of every task in the program.

- The executing task executes for a limited amount of time. On OpenVMS VAX systems, this time is at most the number of seconds (approximate elapsed time) specified by the pragma. Then, if other tasks of the same priority are eligible for execution, the executing task stops executing, and the task that has been waiting the longest is selected for execution.

Notes:

On OpenVMS VAX systems, the amount of scheduling overhead needed to support round-robin task scheduling increases as the value of a time slice decreases. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for the recommended minimum value.

On OpenVMS VAX systems, the DEC Ada predefined package SYSTEM_ RUNTIME_TUNING also has operations that enable time slicing. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* or the package specification for more information.

References: allow 1.6, declarative part 3.9, duration 9.6, fixed point type 3.5.9, library unit 10.1, main program 10.1, pragma 2.8, priority of a task 9.8, static expression 4.9, subprogram 6, task 9

9.8b Pragma Passive (OpenVMS Alpha and Digital UNIX Only)

DEC Ada provides the pragma PASSIVE to direct the compiler to make a particular task passive or to explicitly prevent the compiler from making a task passive.

The form of this pragma is as follows:

```
pragma PASSIVE [(passive_form)];
passive_form => SEMAPHORE | NO
```

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 the containing task should not be optimized.

```
pragma PASSIVE(NO);
```

The pragma is allowed only within a task specification. The specification can be for a task type or for a single task.

A passive task is a compiler optimization of an (unoptimized) Ada task. Passive tasks behave exactly as unoptimized Ada tasks but yield dramatic performance improvements. Not all tasks can be made passive. The best candidates are 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.

If a task does not meet the requirements for a passive task, the task is not made passive.

For more information on the requirements governing passive tasks, see the DEC Ada run-time reference manuals.

References: library unit 10.1, pragma 2.8, task 9

9.9 Task and Entry Attributes

1   For a task object or value T the following attributes are defined:

2   T'CALLABLE   Yields the value FALSE when the execution of the task designated by T is either completed or terminated, or when the task is abnormal. Yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN.

3   T'TERMINATED   Yields the value TRUE if the task designated by T is terminated. Yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN.

4   In addition, the representation attributes STORAGE_SIZE, SIZE, and ADDRESS are defined for a task object T or a task type T (see 13.7.2).

5   The attribute COUNT is defined for an entry E of a task unit T. The entry can be either a single entry or an entry of a family (in either case the name of the single entry or entry family can be either a simple or an expanded name). This attribute is only allowed within the body of T, but excluding within any program unit that is, itself, inner to the body of T.

6   E'COUNT   Yields the number of entry calls presently queued on the entry E (if the attribute is evaluated by the execution of an accept statement for the entry E, the count does not include the calling task). The value of this attribute is of the type *universal_integer*.[20]

---

[20] See also Appendix G, AI-00034.

7 Algorithms interrogating the attribute E′COUNT should take precautions to allow for the increase of the value of this attribute for incoming entry calls, and its decrease, for example with timed entry calls.

8 **References** : abnormal task 9.10, accept statement 9.5, attribute 4.1.4, boolean type 3.5.3, completed task 9.4, designate 9.1, entry 9.5, false boolean value 3.5.3, queue of entry calls 9.5, storage unit 13.7, task 9, task object 9.2, task type 9.1, terminated task 9.4, timed entry call 9.7.3, true boolean value 3.5.3, universal_integer type 3.5.4

9.10 Abort Statements

1 An abort statement causes one or more tasks to become *abnormal*, thus preventing any further rendezvous with such tasks.

2   abort_statement ::= **abort** *task*_name {, *task*_name};

3 The determination of the type of each task name uses the fact that the type of the name is a task type.

4 For the execution of an abort statement, the given task names are evaluated in some order that is not defined by the language. Each named task then becomes abnormal unless it is already terminated; similarly, any task that depends on a named task becomes abnormal unless it is already terminated.

5 Any abnormal task whose execution is suspended at an accept statement, a select statement, or a delay statement becomes completed; any abnormal task whose execution is suspended at an entry call, and that is not yet in a corresponding rendezvous, becomes completed and is removed from the entry queue; any abnormal task that has not yet started its activation becomes completed (and hence also terminated). This completes the execution of the abort statement.[21]

6 The completion of any other abnormal task need not happen before completion of the abort statement. It must happen no later than when the abnormal task reaches a synchronization point that is one of the following: the end of its activation; a point where it causes the activation of another task; an entry call; the start or the end of an accept statement; a select statement; a delay statement; an exception handler; or an abort statement. If a task that calls an entry becomes abnormal while in a rendezvous, its termination does not take place before the completion of the rendezvous (see 11.5).[22]

---

[21] See also Appendix G, AI-00198.
[22] See also Appendix G, AI-00446.

7   The call of an entry of an abnormal task raises the exception TASKING_
     ERROR at the place of the call. Similarly, the exception TASKING_ERROR
     is raised for any task that has called an entry of an abnormal task, if the
     entry call is still queued or if the rendezvous is not yet finished (whether the
     entry call is an entry call statement, or a conditional or timed entry call); the
     exception is raised no later than the completion of the abnormal task. The
     value of the attribute CALLABLE is FALSE for any task that is abnormal
     (or completed).

8   If the abnormal completion of a task takes place while the task updates a
     variable, then the value of this variable is undefined.

9   **Example:**

     ```
     abort USER, TERMINAL.all, POOL(3);
     ```

     Notes:

10   An abort statement should be used only in extremely severe situations
     requiring unconditional termination. A task is allowed to abort any task,
     including itself.

     The rules for an abort statement permit either an *asynchronous* or a
     synchronous implementation of abnormal task completion. An asynchronous
     implementation causes an abnormal task to become completed at arbitrary
     points in its execution (except where prohibited by the above rules). A
     synchronous implementation causes an abnormal task to become completed
     only at specific points in its execution. These points must include the
     synchronization points listed above.

     DEC Ada uses the synchronous implementation. A means of ensuring the
     completion of an abnormal task at a particular point in a DEC Ada program is
     to insert a delay 0.0 statement at that point.

     For more information on the DEC Ada implementation of the abort statement,
     see the relevant DEC Ada run-time reference manual.

11   **References** : abnormal in rendezvous 11.5, accept statement 9.5, activation 9.3,
     attribute 4.1.4, callable (predefined attribute) 9.9, conditional entry call 9.7.2, delay
     statement 9.6, dependent task 9.4, entry call statement 9.5, evaluation of a name 4.1,
     exception handler 11.2, false boolean value 3.5.3, name 4.1, queue of entry calls 9.5,
     rendezvous 9.5, select statement 9.7, statement 5, task 9, tasking_error exception 11.1,
     terminated task 9.4, timed entry call 9.7.3

9.11 Shared Variables

1   The normal means of communicating values between tasks is by entry calls
and accept statements.

2   If two tasks read or update a *shared* variable (that is, a variable accessible by
both), then neither of them may assume anything about the order in which
the other performs its operations, except at the points where they synchronize.
Two tasks are synchronized at the start and at the end of their rendezvous.
At the start and at the end of its activation, a task is synchronized with the
task that causes this activation. A task that has completed its execution is
synchronized with any other task.

3   For the actions performed by a program that uses shared variables, the
following assumptions can always be made:

4   •   If between two synchronization points of a task, this task reads a
shared variable whose type is a scalar or access type, then the variable
is not updated by any other task at any time between these two points.

5   •   If between two synchronization points of a task, this task updates a
shared variable whose type is a scalar or access type, then the variable
is neither read nor updated by any other task at any time between
these two points.

6   The execution of the program is erroneous if any of these assumptions is
violated.

7   If a given task reads the value of a shared variable, the above assumptions
allow an implementation to maintain local copies of the value (for example, in
registers or in some other form of temporary storage); and for as long as the
given task neither reaches a synchronization point nor updates the value of the
shared variable, the above assumptions imply that, for the given task, reading
a local copy is equivalent to reading the shared variable itself.

8   Similarly, if a given task updates the value of a shared variable, the above
assumptions allow an implementation to maintain a local copy of the value,
and to defer the effective store of the local copy into the shared variable until
a synchronization point, provided that every further read or update of the
variable by the given task is treated as a read or update of the local copy. On
the other hand, an implementation is not allowed to introduce a store, unless
this store would also be executed in the canonical order (see 11.6).

9   The pragma SHARED can be used to specify that every read or update of
    a variable is a synchronization point for that variable; that is, the above
    assumptions always hold for the given variable (but not necessarily for other
    variables). The form of this pragma is as follows:[23]

```
pragma SHARED(variable_simple_name);
```

10 This pragma is allowed only for a variable declared by an object declaration
    and whose type is a scalar or access type; the variable declaration and
    the pragma must both occur (in this order) immediately within the same
    declarative part or package specification; the pragma must appear before any
    occurrence of the name of the variable, other than in an address clause.

11 An implementation must restrict the objects for which the pragma SHARED
    is allowed to objects for which each of direct reading and direct updating is
    implemented as an indivisible operation.

DEC Ada does not allow the pragma SHARED for objects of generic formal
floating point types and types derived therefrom.

On OpenVMS VAX systems, DEC Ada does not allow the pragma SHARED
for objects of floating point types whose representation is not the same as
the representation of the type STANDARD.FLOAT (F_floating or IEEE single
float).

See the DEC Ada run-time reference manuals for more information on the
representation of types and objects and on the use of the pragma SHARED.

In addition to the pragma SHARED, DEC Ada provides the pragma VOLATILE
to allow variables that are subject to asynchronous modification to be specified
as such.

The pragma VOLATILE prevents the compiler from referring to an earlier
read or write of the variable to deduce the variable's current value. Every
read of the variable reads the variable itself, rather than a copy of the variable
located in temporary storage. Likewise, every update of the variable updates
the variable itself, rather than a temporary copy. However, if the variable
is in memory shared by two or more processes, each process may have its
own cached copy of the variable. A write to this kind of variable must be
synchronized by a rendezvous, a machine instruction, or a write to an object
that has been specified with the pragma SHARED.

---

[23] See also Appendix G, AI-00141.

Unlike the pragma SHARED, the pragma VOLATILE does not guarantee indivisible access to the shared variable. In other words, it is possible to read partially updated values of the variable if other synchronization mechanisms (rendezvous, machine instructions, and so on) have not been used. Tasks that share a volatile variable must provide their own means of synchronizing their references. The form of this pragma is as follows:

```
pragma VOLATILE (variable_simple_name)
```

This pragma is allowed only for a variable declared by an object declaration; the variable can be of any type. The variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification; the pragma must appear before any occurrence of the name of the variable, other than in an address clause or in one of the DEC Ada pragmas IMPORT_OBJECT, INTERFACE_NAME, EXPORT_OBJECT, or COMMON_OBJECT. It must not occur in combination with the pragma SHARED.

The variable simple name must not be the result of a renaming declaration. If a variable is specified with the pragma VOLATILE, then any renaming of it, or any of its components, is also volatile.

Example:

```
CONSTANT_FIVE        : constant INTEGER := 5;
VOLATILE_VAR, DUMMY : INTEGER;
pragma VOLATILE (VOLATILE_VAR);

begin
    VOLATILE_VAR := CONSTANT_FIVE;  -- statement 1
     . . .
    DUMMY := VOLATILE_VAR;          -- statement 2
end;
```

In this example, statement 1 represents an update of the variable VOLATILE_VAR, and statement 2 represents a read of the variable VOLATILE_VAR. The pragma VOLATILE indicates to the compiler that the variable VOLATILE_VAR may be subject to asynchronous modification. In other words, it may be read or updated by a parallel task or some other mechanism at unpredictable times. Consequently, the compiler always refers to the variable VOLATILE_VAR itself, rather than to a local copy.

To further illustrate, suppose that another task were to *update* the value of VOLATILE_VAR between statements 1 and 2. Then, the pragma VOLATILE ensures that the value of VOLATILE_VAR used in statement 2 is the value updated by the parallel task and not the value assigned in statement 1.

Suppose, instead, that another task were to *read* the value of VOLATILE_
VAR sometime after statement 1, but before statement 2. Then, the pragma
VOLATILE ensures that the value of VOLATILE_VAR used by that task is the
value assigned by statement 1.

Notes:

On OpenVMS systems, any variable in DEC Ada that is asynchronously read
or written by an OpenVMS system service must be specified with a pragma
VOLATILE, or the program will be erroneous (see the *OpenVMS System
Services Reference Manual* for information on which system services have
parameters that are asynchronously modified).

If a variable is shared among tasks such that the assumptions about shared
variables given at the beginning of section 9.11 hold, then a pragma VOLATILE
is not needed.

Because of rules about forcing occurrences (see 13.1), a pragma VOLATILE or
a pragma SHARED for an object specified with an address clause must follow
the address clause.

12 **References** : accept statement 9.5, activation 9.3, assignment 5.2, canonical order
11.6, declarative part 3.9, entry call statement 9.5, erroneous 1.6, global 8.1, package
specification 7.1, pragma 2.8, read a value 6.2, rendezvous 9.5, simple name 3.1 4.1,
task 9, type 3.3, update a value 6.2, variable 3.2.1

allow 1.6, component 3.3, f_floating representation 3.5.7, name 4.1, object declaration
3.2.1, package standard C, renaming declaration 8.5

9.12 Example of Tasking

1 The following example defines a buffering task to smooth variations between
the speed of output of a producing task and the speed of input of some
consuming task. For instance, the producing task may contain the statements

2
```
loop
    --  produce the next character CHAR
    BUFFER.WRITE(CHAR);
    exit when CHAR = ASCII.EOT;
end loop;
```

3 and the consuming task may contain the statements

4
```
loop
    BUFFER.READ(CHAR);
    --  consume the character CHAR
    exit when CHAR = ASCII.EOT;
end loop;
```

5   The buffering task contains an internal pool of characters processed in a round-robin fashion. The pool has two indices, an IN_INDEX denoting the space for the next input character and an OUT_INDEX denoting the space for the next output character.

6
```
task BUFFER is
    entry READ (C : out CHARACTER);
    entry WRITE(C : in  CHARACTER);
end;

task body BUFFER is
    POOL_SIZE : constant INTEGER := 100;
    POOL      : array(1 .. POOL_SIZE) of CHARACTER;
    COUNT     : INTEGER range 0 .. POOL_SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
begin
    loop
        select
            when COUNT < POOL_SIZE =>
                accept WRITE(C : in CHARACTER) do
                    POOL(IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
                COUNT    := COUNT + 1;
        or when COUNT > 0 =>
                accept READ(C : out CHARACTER) do
                    C := POOL(OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
                COUNT     := COUNT - 1;
        or
            terminate;
        end select;
    end loop;
end BUFFER;
```

9.12a Task Entries and OpenVMS Asynchronous System Traps (OpenVMS Systems Only)

An asynchronous system trap (AST) is a call made by the OpenVMS operating system in response to certain events detected or caused by the operating system. In general, an AST occurs upon successful completion of a requested system service if the appropriate parameters have been specified as part of the system service call. ASTs can be handled in DEC Ada with the pragma AST_ENTRY and the AST_ENTRY attribute.

OpenVMS system services that deliver ASTs can be called in DEC Ada with the subprograms provided in the DEC Ada package STARLET. (The package TASKING_SERVICES also provides subprograms for calling those system services that generate ASTs, but no AST handler can be provided by the Ada program calling these operations.) For information on which system services provide an AST option and for a detailed description of ASTs, see the *OpenVMS Programming Concepts Manual*. For information on using AST system services in an Ada program, see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*.

The pragma AST_ENTRY identifies an Ada task entry as one that can subsequently be called to handle an AST. This pragma must be used in combination with the AST_ENTRY attribute and is allowed only in the same task type specification (or single task) as the entry to which it applies. The form of this pragma is as follows:

```
pragma AST_ENTRY (entry_simple_name);
```

The entry simple name must denote a unique entry declared with either zero or one formal parameter. It may not denote an entry family or a member of an entry family. At most one such pragma may be given for any one entry.

When the AST occurs, an entry call is queued to the given entry and the AST is dismissed. Task execution then proceeds according to normal Ada rules. In particular, the rendezvous that results from the AST does *not* execute at AST level.

If the entry has a formal parameter, the parameter must be of a discrete, address, or access type, and the parameter must be of mode **in**. When the AST occurs and the entry is called, the formal parameter receives the value of the astprm parameter provided by the system service.

To connect OpenVMS ASTs with Ada task entries, DEC Ada provides the following attribute, where E is the name of a single entry of a task:

E'AST_ENTRY   Yields a value of the predefined type AST_HANDLER (declared in the predefined package SYSTEM) that enables the given entry, E, to be called when an AST occurs. If the name to which the attribute applies has not been specified with the pragma AST_ENTRY, the attribute returns the value SYSTEM.NO_AST_HANDLER, and no AST occurs. If the entry is for a task that is not callable (T'CALLABLE is false), the exception PROGRAM_ERROR is raised. If an AST occurs for an entry of a task that is terminated, the program is erroneous.

E'AST_ENTRY is typically used and generally only useful as an actual parameter corresponding to the astadr formal parameter of an OpenVMS system service that provides an AST option.

Example:

```
with TEXT_IO; use TEXT_IO;
with STARLET; use STARLET;
with CONDITION_HANDLING; use CONDITION_HANDLING;
procedure AST_EXAMPLE is
   RETURN_STATUS: COND_VALUE_TYPE;
   IO_CHANNEL:    CHANNEL_TYPE;
   FUNCTION_CODE: FUNCTION_CODE_TYPE;

   task AST_HANDLER is
      entry RECEIVE_AST (ASTPRM: in INTEGER);
      pragma AST_ENTRY (RECEIVE_AST);
   end AST_HANDLER;

   task body AST_HANDLER is
   begin
      loop
         select
            accept RECEIVE_AST(ASTPRM: in INTEGER) do
               if ASTPRM = 3 then
                  PUT_LINE("Received the expected AST parameter");
               end if;
            end;
         or
            terminate;
         end select;
      end loop;
   end AST_HANDLER;
```

```
begin
  --
  -- Code to initialize the IO_CHANNEL and FUNCTION_CODE
  -- variables
  --
  -- Call the OpenVMS SYS$QIO system service using the
  -- DEC Ada package STARLET interface
  --
  STARLET.QIO(STATUS => RETURN_STATUS,
              CHAN  => IO_CHANNEL,
              FUNC  => FUNCTION_CODE,
              ASTADR => AST_HANDLER.RECEIVE_AST'AST_ENTRY,
              ASTPRM => 3);
  . . .
end AST_EXAMPLE;
  . . .
```

Note:

Because it depends on the DEC Ada predefined type SYSTEM.AST_
HANDLER, the AST_ENTRY attribute can only be used in a compilation
unit to which the predefined package SYSTEM applies.

References: access type 3.8, actual parameter 6.4 6.4.1, address type 13.7 13.7a.1,
allow 1.6, attribute 4.1.4, callable (predefined attribute) 9.9, discrete type 3.5, entry
9.5, entry call 9.5 9.7.2 9.7.3, entry family 9.5, entry name 9.5, erroneous 1.6, formal
parameter 6.1 6.2, import pragma 13.9a, mode in 6.2, package system 13.7, pragma
import_valued_procedure 13.9a.1.1, pragma interface 13.9, procedure 6, program_error
exception 11.1, rendezvous 9.5, subprogram 6, system.ast_handler 13.7b.3, system.no_
ast_handler 13.7b.3, task specification 9.1, task type 9.2

10

Program Structure and Compilation Issues

1   The overall structure of programs and the facilities for separate compilation are described in this chapter. A program is a collection of one or more compilation units submitted to a compiler in one or more compilations. Each compilation unit specifies the separate compilation of a construct which can be a subprogram declaration or body, a package declaration or body, a generic declaration or body, or a generic instantiation. Alternatively this construct can be a subunit, in which case it includes the body of a subprogram, package, task unit, or generic unit declared within another compilation unit.

2   **References:** compilation 10.1, compilation unit 10.1, generic body 12.2, generic declaration 12.1, generic instantiation 12.3, package body 7.1, package declaration 7.1, subprogram body 6.3, subprogram declaration 6.1, subunit 10.2, task body 9.1, task unit 9

10.1 Compilation Units—Library Units

1   The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation units.

2
```
compilation ::= {compilation_unit}

compilation_unit ::=
    context_clause library_unit | context_clause secondary_unit

library_unit ::=
    subprogram_declaration | package_declaration
  | generic_declaration    | generic_instantiation
  | subprogram_body

secondary_unit ::= library_unit_body | subunit

library_unit_body ::= subprogram_body | package_body
```

3   The compilation units of a program are said to belong to a *program library.* A compilation unit defines either a library unit or a secondary unit. A secondary unit is either the separately compiled proper body of a library unit, or a subunit of another compilation unit. The designator of a separately compiled subprogram (whether a library unit or a subunit) must be an identifier. Within

a program library the simple names of all library units must be distinct identifiers.[1]

4   The effect of compiling a library unit is to define (or redefine) this unit as one that belongs to the program library. For the visibility rules, each library unit acts as a declaration that occurs immediately within the package STANDARD.

5   The effect of compiling a secondary unit is to define the body of a library unit, or in the case of a subunit, to define the proper body of a program unit that is declared within another compilation unit.

6   A subprogram body given in a compilation unit is interpreted as a secondary unit if the program library already contains a library unit that is a subprogram with the same name; it is otherwise interpreted both as a library unit and as the corresponding library unit body (that is, as a secondary unit).[2]

7   The compilation units of a compilation are compiled in the given order. A pragma that applies to the whole of a compilation must appear before the first compilation unit of that compilation.

8   A subprogram that is a library unit can be used as a *main program* in the usual sense. Each main program acts as if called by some environment task; the means by which this execution is initiated are not prescribed by the language definition. An implementation may impose certain requirements on the parameters and on the result, if any, of a main program (these requirements must be stated in Appendix F). In any case, every implementation is required to allow, at least, main programs that are parameterless procedures, and every main program must be a subprogram that is a library unit.

DEC Ada permits a library unit to be used as a main program under the following conditions:

• If it is a procedure with no formal parameters.

   On OpenVMS systems, the status returned to the OpenVMS environment upon normal completion of the procedure is the value 1.

   On Digital UNIX systems, the status returned to the Digital UNIX environment upon normal completion of the procedure is the value 0.

   On Digital UNIX systems, the status returned to the Digital UNIX environment when unhandled exceptions have been raised is the value 42.

---

[1] See also Appendix G, AI-00418.
[2] See also Appendix G, AI-00199, AI-00225, and AI-00266.

- If it is a function with no formal parameters whose returned value is of a discrete type. In this case, the status returned to the operating-system environment upon normal completion of the function is the function value.

- If it is a procedure declared with the pragma EXPORT_VALUED_ PROCEDURE, and it has one formal **out** parameter that is of a discrete type. In this case, the status returned to the operating-system environment upon normal completion of the procedure is the value of the first (and only) parameter.

When a main function or a main procedure declared with the pragma EXPORT_VALUED_PROCEDURE returns a discrete value whose size is less than 32 bits (on OpenVMS VAX systems) or 64 bits (on Alpha systems), the value is zero- or sign-extended as appropriate.

Notes:

9　A simple program may consist of a single compilation unit. A compilation need not have any compilation units; for example, its text can consist of pragmas.

10　The designator of a library function cannot be an operator symbol, but a renaming declaration is allowed to rename a library function as an operator. Two library subprograms must have distinct simple names and hence cannot overload each other. However, renaming declarations are allowed to define overloaded names for such subprograms, and a locally declared subprogram is allowed to overload a library subprogram. The expanded name STANDARD.L can be used for a library unit L (unless the name STANDARD is hidden) since library units act as declarations that occur immediately within the package STANDARD.

11　**References:** allow 1.6, context clause 10.1.1, declaration 3.1, designator 6.1, environment 10.4, generic declaration 12.1, generic instantiation 12.3, hiding 8.3, identifier 2.3, library unit 10.5, local declaration 8.1, must 1.6, name 4.1, occur immediately within 8.1, operator 4.5, operator symbol 6.1, overloading 6.6 8.7, package body 7.1, package declaration 7.1, parameter of a subprogram 6.2, pragma 2.8, procedure 6.1, program unit 6, proper body 3.9, renaming declaration 8.5, simple name 4.1, standard package 8.6, subprogram 6, subprogram body 6.3, subprogram declaration 6.1, subunit 10.2, task 9, visibility 8.3

discrete type 3.5, formal parameter 6.1, function 6.5, pragma export_valued_procedure 13.9a.1.3

10.1.1 Context Clauses—With Clauses

1   A context clause is used to specify the library units whose names are needed within a compilation unit.

2
```
context_clause ::= {with_clause {use_clause}}

with_clause ::= with unit_simple_name {, unit_simple_name};
```

3   The names that appear in a context clause must be the simple names of library units. The simple name of any library unit is allowed within a with clause. The only names allowed in a use clause of a context clause are the simple names of library packages mentioned by previous with clauses of the context clause. A simple name declared by a renaming declaration is not allowed in a context clause.

4   The with clauses and use clauses of the context clause of a library unit *apply* to this library unit and also to the secondary unit that defines the corresponding body (whether such a clause is repeated or not for this unit). Similarly, the with clauses and use clauses of the context clause of a compilation unit *apply* to this unit and also to its subunits, if any.[3]

5   If a library unit is named by a with clause that applies to a compilation unit, then this library unit is directly visible within the compilation unit, except where hidden; the library unit is visible as if declared immediately within the package STANDARD (see 8.6).

6   Dependences among compilation units are defined by with clauses; that is, a compilation unit that mentions other library units in its with clauses *depends* on those library units. These dependences between units are taken into account for the determination of the allowed order of compilation (and recompilation) of compilation units, as explained in section 10.3, and for the determination of the allowed order of elaboration of compilation units, as explained in section 10.5.

Notes:

7   A library unit named by a with clause of a compilation unit is visible (except where hidden) within the compilation unit and hence can be used as a corresponding program unit. Thus within the compilation unit, the name of a library package can be given in use clauses and can be used to form expanded names; a library subprogram can be called; and instances of a library generic unit can be declared.

8   The rules given for with clauses are such that the same effect is obtained whether the name of a library unit is mentioned once or more than once by the applicable with clauses, or even within a given with clause.

---

[3] See also Appendix G, AI-00226.

Example 1: A main program:

9   The following is an example of a main program consisting of a single
compilation unit: a procedure for printing the real roots of a quadratic
equation. The predefined package TEXT_IO and a user-defined package
REAL_OPERATIONS (containing the definition of the type REAL and of
the packages REAL_IO and REAL_FUNCTIONS) are assumed to be already
present in the program library. Such packages may be used by other main
programs.

```ada
10   with TEXT_IO, REAL_OPERATIONS; use REAL_OPERATIONS;
     procedure QUADRATIC_EQUATION is
        A, B, C, D : REAL;
        use REAL_IO,          -- achieves direct visibility of
                              -- GET and PUT for REAL

            TEXT_IO,          -- achieves direct visibility of
                              -- PUT for strings and of NEW_LINE

            REAL_FUNCTIONS;   -- achieves direct visibility of SQRT
     begin
        GET(A); GET(B); GET(C);
        D := B**2 - 4.0*A*C;
        if D < 0.0 then
           PUT("Imaginary Roots.");
        else
           PUT("Real Roots : X1 = ");
           PUT((-B - SQRT(D))/(2.0*A)); PUT(" X2 = ");
           PUT((-B + SQRT(D))/(2.0*A));
        end if;
        NEW_LINE;
     end QUADRATIC_EQUATION;
```

Notes on the example:

11   The with clauses of a compilation unit need only mention the names of those
library subprograms and packages whose visibility is actually necessary within
the unit. They need not (and should not) mention other library units that are
used in turn by some of the units named in the with clauses, unless these
other library units are also used directly by the current compilation unit. For
example, the body of the package REAL_OPERATIONS may need elementary
operations provided by other packages. The latter packages should not be
named by the with clause of QUADRATIC_EQUATION since these elementary
operations are not directly called within its body.

12   **References:** allow 1.6, compilation unit 10.1, direct visibility 8.3, elaboration 3.9,
generic body 12.2, generic unit 12.1, hiding 8.3, instance 12.3, library unit 10.1, main
program 10.1, must 1.6, name 4.1, package 7, package body 7.1, package declaration
7.1, procedure 6.1, program unit 6, secondary unit 10.1, simple name 4.1, standard
predefined package 8.6, subprogram body 6.3, subprogram declaration 6.1, subunit
10.2, type 3.3, use clause 8.4, visibility 8.3

10.1.2 Examples of Compilation Units

1   A compilation unit can be split into a number of compilation units. For
    example, consider the following program.

2   **procedure** PROCESSOR **is**

```
    SMALL : constant := 20;
    TOTAL : INTEGER  := 0;

    package STOCK is
       LIMIT : constant := 1000;
       TABLE : array (1 .. LIMIT) of INTEGER;
       procedure RESTART;
    end STOCK;

    package body STOCK is
       procedure RESTART is
       begin
          for N in 1 .. LIMIT loop
             TABLE(N) := N;
          end loop;
       end;
    begin
       RESTART;
    end STOCK;

    procedure UPDATE(X : INTEGER) is
       use STOCK;
    begin
       ...
       TABLE(X) := TABLE(X) + SMALL;
       ...
    end UPDATE;

begin
    ...
    STOCK.RESTART;  -- reinitializes TABLE
    ...
end PROCESSOR;
```

3   The following three compilation units define a program with an effect
    equivalent to the above example (the broken lines between compilation
    units serve to remind the reader that these units need not be contiguous texts).

4   **Example 2: Several compilation units:**

5   ```
    package STOCK is
        LIMIT : constant := 1000;
        TABLE : array (1 .. LIMIT) of INTEGER;
        procedure RESTART;
    end STOCK;
    ```

    ----

6   ```
     package body STOCK is
        procedure RESTART is
        begin
           for N in 1 .. LIMIT loop
              TABLE(N) := N;
           end loop;
        end;
    begin
        RESTART;
    end STOCK;
    ```

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

7   ```
    with STOCK;
    procedure PROCESSOR is

        SMALL : constant := 20;
        TOTAL : INTEGER  := 0;

        procedure UPDATE(X : INTEGER) is
           use STOCK;
        begin
           ...
           TABLE(X) := TABLE(X) + SMALL;
           ...
        end UPDATE;
    begin
        ...
        STOCK.RESTART;  --  reinitializes TABLE
        ...
    end PROCESSOR;
    ```

8   Note that in the latter version, the package STOCK has no visibility of outer
    identifiers other than the predefined identifiers (of the package STANDARD).
    In particular, STOCK does not use any identifier declared in PROCESSOR
    such as SMALL or TOTAL; otherwise STOCK could not have been extracted
    from PROCESSOR in the above manner. The procedure PROCESSOR, on
    the other hand, depends on STOCK and mentions this package in a with
    clause. This permits the inner occurrences of STOCK in the expanded name
    STOCK.RESTART and in the use clause.

9   These three compilation units can be submitted in one or more compilations. For example, it is possible to submit the package specification and the package body together and in this order in a single compilation.

10 **References:** compilation unit 10.1, declaration 3.1, identifier 2.3, package 7, package body 7.1, package specification 7.1, program 10, standard package 8.6, use clause 8.4, visibility 8.3, with clause 10.1.1

10.2 Subunits of Compilation Units

1   A subunit is used for the separate compilation of the proper body of a program unit declared within another compilation unit. This method of splitting a program permits hierarchical program development.

2
```
body_stub ::=
      subprogram_specification is separate;
    | package body package_simple_name is separate;
    | task body task_simple_name is separate;

subunit ::=
      separate (parent_unit_name) proper_body
```

3   A body stub is only allowed as the body of a program unit (a subprogram, a package, a task unit, or a generic unit) if the body stub occurs immediately within either the specification of a library package or the declarative part of another compilation unit.[4]

4   If the body of a program unit is a body stub, a separately compiled subunit containing the corresponding proper body is required. In the case of a subprogram, the subprogram specifications given in the proper body and in the body stub must conform (see 6.3.1).

5   Each subunit mentions the name of its *parent unit*, that is, the compilation unit where the corresponding body stub is given. If the parent unit is a library unit, it is called the *ancestor* library unit. If the parent unit is itself a subunit, the parent unit name must be given in full as an expanded name, starting with the simple name of the ancestor library unit. The simple names of all subunits that have the same ancestor library unit must be distinct identifiers.[5]

6   Visibility within the proper body of a subunit is the visibility that would be obtained at the place of the corresponding body stub (within the parent unit) if the with clauses and use clauses of the subunit were appended to the context clause of the parent unit. If the parent unit is itself a subunit, then the same rule is used to define the visibility within the proper body of the parent unit.

---

[4]   See also Appendix G, AI-00035.
[5]   See also Appendix G, AI-00289.

7   The effect of the elaboration of a body stub is to elaborate the proper body of
     the subunit.

     Notes:

8   Two subunits of different library units in the same program library need not
     have distinct identifiers. In any case, their full expanded names are distinct,
     since the simple names of library units are distinct and since the simple
     names of all subunits that have a given library unit as ancestor unit are also
     distinct. By means of renaming declarations, overloaded subprogram names
     that rename (distinct) subunits can be introduced.

9   A library unit that is named by the with clause of a subunit can be hidden by a
     declaration (with the same identifier) given in the proper body of the subunit.
     Moreover, such a library unit can even be hidden by a declaration given within
     a parent unit since a library unit acts as if declared in STANDARD; this
     however does not affect the interpretation of the with clauses themselves, since
     only names of library units can appear in with clauses.

10   **References:** compilation unit 10.1, conform 6.3.1, context clause 10.1.1, declaration
     3.1, declarative part 3.9, direct visibility 8.3, elaboration 3.9, expanded name 4.1.3,
     generic body 12.2, generic unit 12, hidden declaration 8.3, identifier 2.3, library unit
     10.1, local declaration 8.1, name 4.1, occur immediately within 8.1, overloading 8.3,
     package 7, package body 7.1, package specification 7.1, program 10, program unit 6,
     proper body 3.9, renaming declaration 8.5, separate compilation 10.1, simple name 4.1,
     subprogram 6, subprogram body 6.3, subprogram specification 6.1, task 9, task body
     9.1, task unit 9.1, use clause 8.4, visibility 8.3, with clause 10.1.1

10.2.1 Examples of Subunits

1   The procedure TOP is first written as a compilation unit without subunits.

2
```
with TEXT_IO;
procedure TOP is

    type REAL is digits 10;
    R, S : REAL := 1.0;

    package FACILITY is
       PI : constant := 3.14159_26536;
       function  F(X : REAL) return REAL;
       procedure G(Y, Z : REAL);
    end FACILITY;

    package body FACILITY is
       --  some local declarations followed by

       function F(X : REAL) return REAL is
       begin
          --  sequence of statements of F
          ...
       end F;
```

```
        procedure G(Y, Z : REAL) is
            --  local procedures using TEXT_IO
            ...
        begin
            --  sequence of statements of G
            ...
        end G;
    end FACILITY;

    procedure TRANSFORM(U : in out REAL) is
        use FACILITY;
    begin
        U := F(U);
        ...
    end TRANSFORM;
begin -- TOP
    TRANSFORM(R);
    ...
    FACILITY.G(R, S);
end TOP;
```

3   The body of the package FACILITY and that of the procedure TRANSFORM can be made into separate subunits of TOP. Similarly, the body of the procedure G can be made into a subunit of FACILITY as follows.

4   **Example 3:**

5
```
procedure TOP is

    type REAL is digits 10;
    R, S : REAL := 1.0;

    package FACILITY is
        PI : constant := 3.14159_26536;
        function  F(X : REAL) return REAL;
        procedure G(Y, Z : REAL);
    end FACILITY;

    package body FACILITY is separate;        --  stub of FACILITY
    procedure TRANSFORM(U : in out REAL) is separate;
                                              --  stub of TRANSFORM

begin  --  TOP
    TRANSFORM(R);
    ...
    FACILITY.G(R, S);
end TOP;
```

```
                  ------------------------------------------------
6       separate (TOP)
        procedure TRANSFORM(U : in out REAL) is
           use FACILITY;
        begin
           U := F(U);
           ...
        end TRANSFORM;

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

7       separate (TOP)
        package body FACILITY is
           --  some local declarations followed by

           function F(X : REAL) return REAL is
           begin
              --  sequence of statements of F
              ...
           end F;

           procedure G(Y, Z : REAL) is separate;    -- stub of G
        end FACILITY;

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

8       with TEXT_IO;
        separate (TOP.FACILITY)  --  full name of FACILITY
        procedure G(Y, Z : REAL) is
           --  local procedures using TEXT_IO
           ...
        begin
           --  sequence of statements of G
           ...
        end G;
```

9   In the above example TRANSFORM and FACILITY are subunits of TOP, and
    G is a subunit of FACILITY. The visibility in the split version is the same as in
    the initial version except for one change: since TEXT_IO is only used within G,
    the corresponding with clause is written for G instead of for TOP. Apart from
    this change, the same identifiers are visible at corresponding program points in
    the two versions. For example, all of the following are (directly) visible within
    the proper body of the subunit G: the procedure TOP, the type REAL, the
    variables R and S, the package FACILITY and the contained named number PI
    and subprograms F and G.

10 **References:** body stub 10.2, compilation unit 10.1, identifier 2.3, local declaration
    8.1, named number 3.2, package 7, package body 7.1, procedure 6, procedure body 6.3,
    proper body 3.9, subprogram 6, type 3.3, variable 3.2.1, visibility 8.3, with clause 10.1.1

10.3 Order of Compilation

1   The rules defining the order in which units can be compiled are direct consequences of the visibility rules and, in particular, of the fact that any library unit that is mentioned by the context clause of a compilation unit is visible in the compilation unit.

2   A compilation unit must be compiled after all library units named by its context clause. A secondary unit that is a subprogram or package body must be compiled after the corresponding library unit. Any subunit of a parent compilation unit must be compiled after the parent compilation unit.

3   If any error is detected while attempting to compile a compilation unit, then the attempted compilation is rejected and it has no effect whatsoever on the program library; the same holds for recompilations (no compilation unit can become obsolete because of such a recompilation).[6]

4   The order in which the compilation units of a program are compiled must be consistent with the partial ordering defined by the above rules.

5   Similar rules apply for recompilations. A compilation unit is potentially affected by a change in any library unit named by its context clause. A secondary unit is potentially affected by a change in the corresponding library unit. The subunits of a parent compilation unit are potentially affected by a change of the parent compilation unit. If a compilation unit is successfully recompiled, the compilation units potentially affected by this change are obsolete and must be recompiled unless they are no longer needed. An implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change.

6   The subunits of a unit can be recompiled without affecting the unit itself. Similarly, changes in a subprogram or package body do not affect other compilation units (apart from the subunits of the body) since these compilation units only have access to the subprogram or package specification. An implementation is only allowed to deviate from this rule for inline inclusions, for certain compiler optimizations, and for certain implementations of generic program units, as described below.[7]

---

[6]   See also Appendix G, AI-00261.
[7]   See also Appendix G, AI-00408.

7 • If a pragma INLINE is applied to a subprogram declaration given in a package specification, inline inclusion will only be achieved if the package body is compiled before units calling the subprogram. In such a case, inline inclusion creates a *dependence* of the calling unit on the package body, and the compiler must recognize this dependence when deciding on the need for recompilation. If a calling unit is compiled before the package body, the pragma may be ignored by the compiler for such calls (a warning that inline inclusion was not achieved may be issued). Similar considerations apply to a separately compiled subprogram for which an INLINE pragma is specified.[8]

• In DEC Ada, if a pragma INLINE_GENERIC is applied to a generic instantiation, either by naming the instantiation or by naming the generic declaration from which the instantiation was derived, inline inclusion will only be achieved if the corresponding generic body is compiled before the instantiation. In such a case, inline inclusion creates a *dependence* of the instantiation on the generic body, and the compiler recognizes this dependence when deciding on the need for recompilation. If an instantiation is compiled before the generic body, the pragma is ignored, and a diagnostic message that inline inclusion was not achieved is issued.

8 • For optimization purposes, an implementation may compile several units of a given compilation in a way that creates further dependences among these compilation units. The compiler must then take these dependences into account when deciding on the need for recompilations.

9 • An implementation may require that a generic declaration and the corresponding proper body be part of the same compilation, whether the generic unit is itself separately compiled or is local to another compilation unit. An implementation may also require that subunits of a generic unit be part of the same compilation.[9]

DEC Ada does not require that a generic declaration and the corresponding proper body be part of the same compilation, nor does DEC Ada require that the subunits of a generic unit be part of the same compilation. The instantiation of a generic declaration before the corresponding body is available results in an incomplete compilation. See the DEC Ada program development manuals for information on compiling and recompiling DEC Ada compilation units.

---

[8] See also Appendix G, AI-00200.
[9] See also Appendix G, AI-00257.

10   **Examples of Compilation Order:**

11   (a)   In example 1 (see 10.1.1): The procedure QUADRATIC_EQUATION
                must be compiled after the library packages TEXT_IO and REAL_
                OPERATIONS since they appear in its with clause.

12   (b)   In example 2 (see 10.1.2): The package body STOCK must be compiled
                after the corresponding package specification.

13   (c)   In example 2 (see 10.1.2): The specification of the package STOCK
                must be compiled before the procedure PROCESSOR. On the other
                hand, the procedure PROCESSOR can be compiled either before or
                after the package body STOCK.

14   (d)   In example 3 (see 10.2.1): The procedure G must be compiled after
                the package TEXT_IO since this package is named by the with clause
                of G. On the other hand, TEXT_IO can be compiled either before or
                after TOP.

15   (e)   In example 3 (see 10.2.1): The subunits TRANSFORM and FACILITY
                must be compiled after the main program TOP. Similarly, the subunit
                G must be compiled after its parent unit FACILITY.

Notes:

16   For library packages, it follows from the recompilation rules that a package
        body is made obsolete by the recompilation of the corresponding specification.
        If the new package specification is such that a package body is not required
        (that is, if the package specification does not contain the declaration of a
        program unit), then the recompilation of a body for this package is not
        required. In any case, the obsolete package body must not be used and can
        therefore be deleted from the program library.

17   **References:** compilation 10.1, compilation unit 10.1, context clause 10.1.1,
        elaboration 3.9, generic body 12.2, generic declaration 12.1, generic unit 12, library
        unit 10.1, local declaration 8.1, name 4.1, package 7, package body 7.1, package
        specification 7.1, parent unit 10.2, pragma inline 6.3.2, procedure 6.1, procedure body
        6.3, proper body 3.9, secondary unit 10.1, subprogram body 6.3, subprogram declaration
        6.1, subprogram specification 6.1, subunit 10.2, type 3.3, variable 3.2.1, visibility 8.3,
        with clause 10.1.1

        generic body 12.2, instantiation 12.3

10.4 The Program Library

1   Compilers are required to enforce the language rules in the same manner
for a program consisting of several compilation units (and subunits) as for
a program submitted as a single compilation. Consequently, a library file
containing information on the compilation units of the program library must be
maintained by the compiler or compiling environment. This information may
include symbol tables and other information pertaining to the order of previous
compilations.

2   A normal submission to the compiler consists of the compilation unit(s) and the
library file. The latter is used for checks and is updated for each compilation
unit successfully compiled.

Notes:

3   A single program library is implied for the compilation units of a compilation.
The possible existence of different program libraries and the means by which
they are named are not concerns of the language definition; they are concerns
of the programming environment.

4   There should be commands for creating the program library of a given program
or of a given family of programs. These commands may permit the reuse
of units of other program libraries. Finally, there should be commands for
interrogating the status of the units of a program library. The form of these
commands is not specified by the language definition.

DEC Ada program library management, as well as the commands for creating
program libraries and for determining the status of the units of a program
library, is described in the DEC Ada program development manuals.

5   **References:** compilation unit 10.1, context clause 10.1.1, order of compilation 10.3,
program 10.1, program library 10.1, subunit 10.2, use clause 8.4, with clause 10.1.1

10.5 Elaboration of Library Units

1   Before the execution of a main program, all library units needed by the main
program are elaborated, as well as the corresponding library unit bodies, if
any. The library units needed by the main program are: those named by
with clauses applicable to the main program, to its body, and to its subunits;
those named by with clauses applicable to these library units themselves, to
the corresponding library unit bodies, and to their subunits; and so on, in a
transitive manner.[10]

---

[10] See also Appendix G, AI-00158.

2   The elaboration of these library units and of the corresponding library unit bodies is performed in an order consistent with the partial ordering defined by the with clauses (see 10.3). In addition, a library unit mentioned by the context clause of a subunit must be elaborated before the body of the ancestor library unit of the subunit.[11]

3   An order of elaboration that is consistent with this partial ordering does not always ensure that each library unit body is elaborated before any other compilation unit whose elaboration necessitates that the library unit body be already elaborated. If the prior elaboration of library unit bodies is needed, this can be requested by a pragma ELABORATE. The form of this pragma is as follows:

```
pragma ELABORATE (library_unit_simple_name
                   {, library_unit_simple_name});
```

4   These pragmas are only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument of such a pragma must be the simple name of a library unit mentioned by the context clause, and this library unit must have a library unit body. Such a pragma specifies that the library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit.[12]

5   The program is illegal if no consistent order can be found (that is, if a circularity exists). The elaboration of the compilation units of the program is performed in some order that is otherwise not defined by the language.

6   **References:** allow 1.6, argument of a pragma 2.8, compilation unit 10.1, context clause 10.1.1, dependence between compilation units 10.3, elaboration 3.9, illegal 1.6, in some order 1.6, library unit 10.1, name 4.1, main program 10.1, pragma 2.8, secondary unit 10.1, separate compilation 10.1, simple name 4.1, subunit 10.2, with clause 10.1.1

---

[11] See also Appendix G, AI-00354.
[12] See also Appendix G, AI-00236, AI-00298, and AI-00355.

10.6 Program Optimization

1   Optimization of the elaboration of declarations and the execution of statements may be performed by compilers. In particular, a compiler may be able to optimize a program by evaluating certain expressions, in addition to those that are static expressions. Should one of these expressions, whether static or not, be such that an exception would be raised by its evaluation, then the code in that path of the program can be replaced by code to raise the exception; the same holds for exceptions raised by the evaluation of names and simple expressions. (See also section 11.6.)

2   A compiler may find that some statements or subprograms will never be executed, for example, if their execution depends on a condition known to be FALSE. The corresponding object machine code can then be omitted. This rule permits the effect of *conditional compilation* within the language.

Note:

3   An expression whose evaluation is known to raise an exception need not represent an error if it occurs in a statement or subprogram that is never executed. The compiler may warn the programmer of a potential error.

4   **References:** condition 5.3, declaration 3.1, elaboration 3.9, evaluation 4.5, exception 11, expression 4.4, false boolean value 3.5.3, program 10, raising of exceptions 11.3, statement 5, static expression 4.9, subprogram 6

11

Exceptions

1   This chapter defines the facilities for dealing with errors or other exceptional situations that arise during program execution. Such a situation is called an *exception.* To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Executing some actions, in response to the arising of an exception, is called *handling* the exception.

2   An exception declaration declares a name for an exception. An exception can be raised by a raise statement, or it can be raised by another statement or operation that *propagates* the exception. When an exception arises, control can be transferred to a user-provided exception handler at the end of a block statement or at the end of the body of a subprogram, package, or task unit.

Note:

In DEC Ada, exceptions are implemented using operating-system-specific and hardware-specific facilities. The DEC Ada run-time reference manuals describe Digital's implementation of Ada exceptions in more detail.

3   **References:** block statement 5.6, error situation 1.6, exception handler 11.2, name 4.1, package body 7.1, propagation of an exception 11.4.1 11.4.2, raise statement 11.3, subprogram body 6.3, task body 9.1

11.1 Exception Declarations

1   An exception declaration declares a name for an exception. The name of an exception can only be used in raise statements, exception handlers, and renaming declarations.

2   `exception_declaration ::= identifier_list : `**`exception`**`;`

3   An exception declaration with several identifiers is equivalent to a sequence of single exception declarations, as explained in section 3.2. Each single exception declaration declares a name for a different exception. In particular, if a generic unit includes an exception declaration, the exception declarations implicitly generated by different instantiations of the generic unit refer to distinct

exceptions (but all have the same identifier). The particular exception denoted by an exception name is determined at compilation time and is the same regardless of how many times the exception declaration is elaborated. Hence, if an exception declaration occurs in a recursive subprogram, the exception name denotes the same exception for all invocations of the recursive subprogram.

4   The following exceptions are predefined in the language; they are raised when the situations described are detected.

5   CONSTRAINT_ERROR   This exception is raised in any of the following situations: upon an attempt to violate a range constraint, an index constraint, or a discriminant constraint; upon an attempt to use a record component that does not exist for the current discriminant values; and upon an attempt to use a selected component, an indexed component, a slice, or an attribute, of an object designated by an access value, if the object does not exist because the access value is null.

In DEC Ada, this exception is also raised by the execution of a predefined numeric operation that cannot deliver a correct result (within the declared accuracy for real types). It is raised for integer overflow, floating point overflow, and integer and floating point division by zero. This exception is not raised by floating point underflow.

On OpenVMS systems, underflow can be handled as an imported OpenVMS condition (see 13.9a.3.1 for information on importing OpenVMS conditions).

6   NUMERIC_ERROR   This exception is raised by the execution of a predefined numeric operation that cannot deliver a correct result (within the declared accuracy for real types); this includes the case where an implementation uses a predefined numeric operation for the execution, evaluation, or elaboration of some construct. The rules given in section 4.5.7 define the cases in which an implementation is not required to raise this

exception when such an error situation arises; see also section 11.6.[1]

DEC Ada raises the exception NUMERIC_ ERROR only when it is explicitly raised with a raise statement. Wherever this standard requires that NUMERIC_ERROR be raised, the exception CONSTRAINT_ERROR is raised instead.[2]

7   PROGRAM_ERROR   This exception is raised upon an attempt to call a subprogram, to activate a task, or to elaborate a generic instantiation, if the body of the corresponding unit has not yet been elaborated. This exception is also raised if the end of a function is reached (see 6.5); or during the execution of a selective wait that has no else part, if this execution determines that all alternatives are closed (see 9.7.1). Finally, depending on the implementation, this exception may be raised upon an attempt to execute an action that is erroneous, and for incorrect order dependences (see 1.6).

In DEC Ada, this exception is raised for some erroneous situations. It is not raised for incorrect order dependences.

8   STORAGE_ERROR   This exception is raised in any of the following situations: when the dynamic storage allocated to a task is exceeded; during the evaluation of an allocator, if the space available for the collection of allocated objects is exhausted; or during the elaboration of a declarative item, or during the execution of a subprogram call, if storage is not sufficient.

9   TASKING_ERROR   This exception is raised when exceptions arise during intertask communication (see 9 and 11.5).

DEC Ada also defines the exception NON_ADA_ERROR (in the package SYSTEM); see sections 11.2 and 13.7a.3.

---

[1]   See also Appendix G, AI-00311, AI-00312, and AI-00387.
[2]   See also Appendix G, AI-00387.

DEC Ada also provides the language-defined package of input-output exceptions: IO_EXCEPTIONS. On OpenVMS systems, DEC Ada provides an additional package of input-output exceptions—AUX_IO_EXCEPTIONS—for use with the DEC Ada relative and indexed input-output packages. The DEC Ada input-output exceptions are described in sections 14.4, 14.5, and 14.5a.

Note:

10   The situations described above can arise without raising the corresponding exceptions, if the pragma SUPPRESS has been used to give permission to omit the corresponding checks (see 11.7).

In addition to the pragma SUPPRESS, DEC Ada provides the pragma SUPPRESS_ALL (see 11.7).

11   **Examples of user-defined exception declarations:**

```
SINGULAR : exception;
ERROR    : exception;
OVERFLOW, UNDERFLOW : exception;
```

12   **References:** access value 3.8, collection 3.8, declaration 3.1, exception 11, exception handler 11.2, generic body 12.2, generic instantiation 12.3, generic unit 12, identifier 2.3, implicit declaration 12.3, instantiation 12.3, name 4.1, object 3.2, raise statement 11.3, real type 3.5.6, record component 3.7, return statement 5.8, subprogram 6, subprogram body 6.3, task 9, task body 9.1

aux_io_exceptions package 14.4, erroneous 1.6, imported VAX condition 13.9a.3.1, indexed input-output 14.2a 14.2a.1 14.2a.4 14.2b.9, indexed_io package 14.2a.5, indexed_mixed_io package 14.2b.10, io_exceptions package 14.4, non_ada_error exception 13.7a.3, package system 13.7, relative input-output 14.2a 14.2a.1 14.2a.2 14.2b.7, relative_io package 14.2a.3, relative_mixed_io package 14.2b.8

13   **Constraint_error exception contexts:** aggregate 4.3.1 4.3.2, allocator 4.8, assignment statement 5.2 5.2.1, constraint 3.3.2, discrete type attribute 3.5.5, discriminant constraint 3.7.2, elaboration of a generic formal parameter 12.3.1 12.3.2 12.3.4 12.3.5, entry index 9.5, exponentiating operator 4.5.6, index constraint 3.6.1, indexed component 4.1.1, logical operator 4.5.1, null access value 3.8, object declaration 3.2.1, parameter association 6.4.1, qualified expression 4.7, range constraint 3.5, selected component 4.1.3, slice 4.1.2, subtype indication 3.3.2, type conversion 4.6

14   **Numeric_error exception contexts:** discrete type attribute 3.5.5, implicit conversion 3.5.4 3.5.6 4.6, numeric operation 3.5.5 3.5.8 3.5.10, operator of a numeric type 4.5 4.5.7

15   **Program_error exception contexts:** collection 3.8, elaboration 3.9, elaboration check 3.9 7.3 9.3 12.2, erroneous 1.6, incorrect order dependence 1.6, leaving a function 6.5, selective wait 9.7.1

16   **Storage_error exception contexts:** allocator 4.8

17   **Tasking_error exception contexts:** abort statement 9.10, entry call 9.5 9.7.2 9.7.3, exceptions during task communication 11.5, task activation 9.3

11.2 Exception Handlers

1   The response to one or more exceptions is specified by an exception handler.

```
2      exception_handler ::=
           when exception_choice {| exception_choice} =>
               sequence_of_statements

       exception_choice ::= exception_name | others
```

3   An exception handler occurs in a construct that is either a block statement or the body of a subprogram, package, task unit, or generic unit. Such a construct will be called a *frame* in this chapter. In each case the syntax of a frame that has exception handlers includes the following part:

```
4      begin
           sequence_of_statements
       exception
           exception_handler
          {exception_handler}
       end
```

5   The exceptions denoted by the exception names given as exception choices of a frame must all be distinct. The exception choice **others** is only allowed for the last exception handler of a frame and as its only exception choice; it stands for all exceptions not listed in previous handlers of the frame, including exceptions whose names are not visible at the place of the exception handler.

6   The exception handlers of a frame handle exceptions that are raised by the execution of the sequence of statements of the frame. The exceptions handled by a given exception handler are those named by the corresponding exception choices.

When non-Ada code is imported in an Ada program (see 13.9a), hardware conditions or exceptions that may be signaled can be handled by an **others** choice in the Ada frame that calls the non-Ada routines. See the DEC Ada run-time reference manuals for more information on handling non-Ada exceptions from Ada programs.

7   **Example:**
```
begin
   --  sequence of statements
exception
   when SINGULAR | NUMERIC_ERROR =>
      PUT(" MATRIX IS SINGULAR ");
   when others =>
      PUT(" FATAL ERROR ");
      raise ERROR;
end;
```

   Note:

8   The same kinds of statement are allowed in the sequence of statements of each
    exception handler as are allowed in the sequence of statements of the frame.
    For example, a return statement is allowed in a handler within a function
    body.

9   **References:** block statement 5.6, declarative part 3.9, exception 11, exception
    handling 11.4, function body 6.3, generic body 12.2, generic unit 12.1, name 4.1,
    package body 7.1, raise statement 11.3, return statement 5.8, sequence of statements
    5.1, statement 5, subprogram body 6.3, task body 9.1, task unit 9 9.1, visibility 8.3

    import pragma 13.9a, non_ada_error exception 13.7a.3, package system 13.7

11.3 Raise Statements

1   A raise statement raises an exception.

2   raise_statement ::= **raise** [*exception*_name];

3   For the execution of a raise statement with an exception name, the named
    exception is raised. A raise statement without an exception name is only
    allowed within an exception handler (but not within the sequence of statements
    of a subprogram, package, task unit, or generic unit, enclosed by the handler);
    it raises again the exception that caused transfer to the innermost enclosing
    handler.

4   **Examples:**
```
raise SINGULAR;
raise NUMERIC_ERROR;  --  explicitly raising a predefined
                      --  exception

raise;                --  only within an exception handler
```

5   **References:** exception 11, generic unit 12, name 4.1, package 7, sequence of
    statements 5.1, subprogram 6, task unit 9

11.4 Exception Handling

1 When an exception is raised, normal program execution is abandoned and control is transferred to an exception handler. The selection of this handler depends on whether the exception is raised during the execution of statements or during the elaboration of declarations.[3]

Note:

DEC Ada allows non-Ada exceptions to be handled using various mechanisms (see 11.2).

On OpenVMS systems, DEC Ada provides the following mechanisms for handling system conditions or exported Ada exceptions:

- The pragma IMPORT_EXCEPTION (see 13.9a.3.1), which allows Ada exception handlers to handle any OpenVMS conditions raised in DEC Ada code or in imported (non-Ada) code.

- The package CONDITION_HANDLING, which allows additional control over OpenVMS condition handling.

- The pragma EXPORT_EXCEPTION (see 13.9a.3.2), which allows handlers written in other languages to handle exported Ada exceptions.

See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for implementation details and for information on using these pragmas and packages.

2 **References:** declaration 3.1, elaboration 3.1 3.9, exception 11, exception handler 11.2, raising of exceptions 11.3, statement 5

11.4.1 Exceptions Raised During the Execution of Statements

1 The handling of an exception raised by the execution of a sequence of statements depends on whether the innermost frame or accept statement that encloses the sequence of statements is a frame or an accept statement. The case where an accept statement is innermost is described in section 11.5. The case where a frame is innermost is presented here.

2 Different actions take place, depending on whether or not this frame has a handler for the exception, and on whether the exception is raised in the sequence of statements of the frame or in that of an exception handler.

---

[3] See also Appendix G, AI-00446.

3   If an exception is raised in the sequence of statements of a frame that has a handler for the exception, execution of the sequence of statements of the frame is abandoned and control is transferred to the exception handler. The execution of the sequence of statements of the handler completes the execution of the frame (or its elaboration if the frame is a package body).[4]

4   If an exception is raised in the sequence of statements of a frame that does not have a handler for the exception, execution of this sequence of statements is abandoned. The next action depends on the nature of the frame:

5   (a)   For a subprogram body, the same exception is raised again at the point of call of the subprogram, unless the subprogram is the main program itself, in which case execution of the main program is abandoned.

6   (b)   For a block statement, the same exception is raised again immediately after the block statement (that is, within the innermost enclosing frame or accept statement).

7   (c)   For a package body that is a declarative item, the same exception is raised again immediately after this declarative item (within the enclosing declarative part). If the package body is that of a subunit, the exception is raised again at the place of the corresponding body stub. If the package is a library unit, execution of the main program is abandoned.

8   (d)   For a task body, the task becomes completed.

9   An exception that is raised again (as in the above cases (a), (b), and (c)) is said to be *propagated*, either by the execution of the subprogram, the execution of the block statement, or the elaboration of the package body. No propagation takes place in the case of a task body. If the frame is a subprogram or a block statement and if it has dependent tasks, the propagation of an exception takes place only after termination of the dependent tasks.

10   Finally, if an exception is raised in the sequence of statements of an exception handler, execution of this sequence of statements is abandoned. Subsequent actions (including propagation, if any) are as in the cases (a) to (d) above, depending on the nature of the frame.

---

[4]   See also Appendix G, AI-00455.

11 **Example:**

```
function FACTORIAL (N : POSITIVE) return FLOAT is
begin
   if N = 1 then
      return 1.0;
   else
      return FLOAT(N) * FACTORIAL(N-1);
   end if;
exception
   when NUMERIC_ERROR => return FLOAT'SAFE_LARGE;
end FACTORIAL;
```

12 If the multiplication raises NUMERIC_ERROR, then FLOAT'SAFE_LARGE is returned by the handler. This value will cause further NUMERIC_ERROR exceptions to be raised by the evaluation of the expression in each of the remaining invocations of the function, so that for large values of N the function will ultimately return the value FLOAT'SAFE_LARGE.

In DEC Ada, the predefined exception CONSTRAINT_ERROR is raised instead of NUMERIC_ERROR (see 11.1).

13 **Example:**

```
procedure P is
   ERROR : exception;
   procedure R;

   procedure Q is
   begin
      R;
      ...              -- error situation (2)
   exception
      ...
      when ERROR =>  -- handler E2
      ...
   end Q;

   procedure R is
   begin
      ...              -- error situation (3)
   end R;

begin
   ...              -- error situation (1)
   Q;
   ...
exception
   ...
   when ERROR =>     -- handler E1
   ...
end P;
```

14 The following situations can arise:

15   (1)   If the exception ERROR is raised in the sequence of statements of the outer procedure P, the handler E1 provided within P is used to complete the execution of P.

16   (2)   If the exception ERROR is raised in the sequence of statements of Q, the handler E2 provided within Q is used to complete the execution of Q. Control will be returned to the point of call of Q upon completion of the handler.

17   (3)   If the exception ERROR is raised in the body of R, called by Q, the execution of R is abandoned and the same exception is raised in the body of Q. The handler E2 is then used to complete the execution of Q, as in situation (2).

18   Note that in the third situation, the exception raised in R results in (indirectly) transferring control to a handler that is part of Q and hence not enclosed by R. Note also that if a handler were provided within R for the exception choice **others**, situation (3) would cause execution of this handler, rather than direct termination of R.

19   Lastly, if ERROR had been declared in R, rather than in P, the handlers E1 and E2 could not provide an explicit handler for ERROR since this identifier would not be visible within the bodies of P and Q. In situation (3), the exception could however be handled in Q by providing a handler for the exception choice **others.**

Notes:

20   The language does not define what happens when the execution of the main program is abandoned after an unhandled exception.

21   The predefined exceptions are those that can be propagated by the basic operations and the predefined operators.

22   The case of a frame that is a generic unit is already covered by the rules for subprogram and package bodies, since the sequence of statements of such a frame is not executed but is the template for the corresponding sequences of statements of the subprograms or packages obtained by generic instantiation.

23   **References:** accept statement 9.5, basic operation 3.3.3, block statement 5.6, body stub 10.2, completion 9.4, declarative item 3.9, declarative part 3.9, dependent task 9.4, elaboration 3.1 3.9, exception 11, exception handler 11.2, frame 11.2, generic instantiation 12.3, generic unit 12, library unit 10.1, main program 10.1, numeric_ error exception 11.1, package 7, package body 7.1, predefined operator 4.5, procedure 6.1, sequence of statements 5.1, statement 5, subprogram 6, subprogram body 6.3, subprogram call 6.4, subunit 10.2, task 9, task body 9.1

11.4.2 Exceptions Raised During the Elaboration of Declarations

1   If an exception is raised during the elaboration of the declarative part of a given frame, this elaboration is abandoned. The next action depends on the nature of the frame:

2   (a)   For a subprogram body, the same exception is raised again at the point of call of the subprogram, unless the subprogram is the main program itself, in which case execution of the main program is abandoned.

3   (b)   For a block statement, the same exception is raised again immediately after the block statement.

4   (c)   For a package body that is a declarative item, the same exception is raised again immediately after this declarative item, in the enclosing declarative part. If the package body is that of a subunit, the exception is raised again at the place of the corresponding body stub. If the package is a library unit, execution of the main program is abandoned.

5   (d)   For a task body, the task becomes completed, and the exception TASKING_ERROR is raised at the point of activation of the task, as explained in section 9.3.

6   Similarly, if an exception is raised during the elaboration of either a package declaration or a task declaration, this elaboration is abandoned; the next action depends on the nature of the declaration.

7   (e)   For a package declaration or a task declaration, that is a declarative item, the exception is raised again immediately after the declarative item in the enclosing declarative part or package specification. For the declaration of a library package, the execution of the main program is abandoned.

8   An exception that is raised again (as in the above cases (a), (b), (c) and (e)) is said to be *propagated*, either by the execution of the subprogram or block statement, or by the elaboration of the package declaration, task declaration or package body.

9 **Example of an exception in the declarative part of a block statement (case (b)):**

```
procedure P is
   ...
begin
   declare
      N : INTEGER := F;  --  the function F may raise ERROR
   begin
      ...
   exception
      when ERROR =>       --  handler E1
   end;
   ...
exception
   when ERROR =>          --  handler E2
end P;

--  if the exception ERROR is raised in the declaration of N,
--  it is handled by E2
```

10 **References:** activation 9.3, block statement 5.6, body stub 10.2, completed task 9.4, declarative item 3.9, declarative part 3.9, elaboration 3.1 3.9, exception 11, frame 11.2, library unit 10.1, main program 10.1, package body 7.1, package declaration 7.1, package specification 7.1, subprogram 6, subprogram body 6.3, subprogram call 6.4, subunit 10.2, task 9, task body 9.1, task declaration 9.1, tasking_error exception 11.1

11.5 Exceptions Raised During Task Communication

1 An exception can be propagated to a task communicating, or attempting to communicate, with another task. An exception can also be propagated to a calling task if the exception is raised during a rendezvous.

2 When a task calls an entry of another task, the exception TASKING_ERROR is raised in the calling task, at the place of the call, if the called task is completed before accepting the entry call or is already completed at the time of the call.

3 A rendezvous can be completed abnormally in two cases:

4   (a) When an exception is raised within an accept statement, but not handled within an inner frame. In this case, the execution of the accept statement is abandoned and the same exception is raised again immediately after the accept statement within the called task; the exception is also propagated to the calling task at the point of the entry call.

5   (b) When the task containing the accept statement is completed abnormally as the result of an abort statement. In this case, the exception TASKING_ERROR is raised in the calling task at the point of the entry call.

6   On the other hand, if a task issuing an entry call becomes abnormal (as the result of an abort statement) no exception is raised in the called task. If the rendezvous has not yet started, the entry call is cancelled. If the rendezvous is in progress, it completes normally, and the called task is unaffected.

7   **References:** abnormal task 9.10, abort statement 9.10, accept statement 9.5, completed task 9.4, entry call 9.5, exception 11, frame 11.2, rendezvous 9.5, task 9, task termination 9.4, tasking_error exception 11.1

11.6 Exceptions and Optimization

1   The purpose of this section is to specify the conditions under which an implementation is allowed to perform certain actions either earlier or later than specified by other rules of the language.

2   In general, when the language rules specify an order for certain actions (the *canonical order*), an implementation may only use an alternative order if it can guarantee that the effect of the program is not changed by the reordering. In particular, no exception should arise for the execution of the reordered program if none arises for the execution of the program in the canonical order. When, on the other hand, the order of certain actions is not defined by the language, any order can be used by the implementation. (For example, the arguments of a predefined operator can be evaluated in any order since the rules given in section 4.5 do not require a specific order of evaluation.)

3   Additional freedom is left to an implementation for reordering actions involving predefined operations that are either predefined operators or basic operations other than assignments. This freedom is left, as defined below, even in the case where the execution of these predefined operations may propagate a (predefined) exception:

4   (a)   For the purpose of establishing whether the same effect is obtained by the execution of certain actions in the canonical and in an alternative order, it can be assumed that none of the predefined operations invoked by these actions propagates a (predefined) exception, provided that the two following requirements are met by the alternative order: first, an operation must not be invoked in the alternative order if it is not invoked in the canonical order; second, for each operation, the innermost enclosing frame or accept statement must be the same in the alternative order as in the canonical order, and the same exception handlers must apply.

5   (b)   Within an expression, the association of operators with operands is specified by the syntax. However, for a sequence of predefined operators of the same precedence level (and in the absence of parentheses imposing a specific association), any association of operators with operands is allowed if it satisfies the following requirement: an integer result must be equal to that given by the canonical left-to-right order; a real result must belong to the result model interval defined for the canonical left-to-right order (see 4.5.7). Such a reordering is allowed even if it may remove an exception, or introduce a further predefined exception.

6   Similarly, additional freedom is left to an implementation for the evaluation of numeric simple expressions. For the evaluation of a predefined operation, an implementation is allowed to use the operation of a type that has a range wider than that of the base type of the operands, provided that this delivers the exact result (or a result within the declared accuracy, in the case of a real type), even if some intermediate results lie outside the range of the base type. The exception NUMERIC_ERROR need not be raised in such a case. In particular, if the numeric expression is an operand of a predefined relational operator, the exception NUMERIC_ERROR need not be raised by the evaluation of the relation, provided that the correct BOOLEAN result is obtained.[5]

7   A predefined operation need not be invoked at all, if its only possible effect is to propagate a predefined exception. Similarly, a predefined operation need not be invoked if the removal of subsequent operations by the above rule renders this invocation ineffective.

Notes:

8   Rule (b) applies to predefined operators but not to the short-circuit control forms.

9   The expression SPEED < 300_000.0 can be replaced by TRUE if the value 300_000.0 lies outside the base type of SPEED, even though the implicit conversion of the numeric literal would raise the exception NUMERIC_ERROR.

---

[5]   See also Appendix G, AI-00267.

10 **Example:**

```
declare
   N : INTEGER;
begin
   N := 0;                -- (1)
   for J in 1 .. 10 loop
      N := N + J**A(K);  -- A and K are global variables
   end loop;
   PUT(N);
exception
   when others => PUT("Some error arose"); PUT(N);
end;
```

11   The evaluation of A(K) may be performed before the loop, and possibly immediately before the assignment statement (1) even if this evaluation can raise an exception. Consequently, within the exception handler, the value of N is either the undefined initial value or a value later assigned. On the other hand, the evaluation of A(K) cannot be moved before **begin** since an exception would then be handled by a different handler. For this reason, the initialization of N in the declaration itself would exclude the possibility of having an undefined initial value of N in the handler.

12   **References:** accept statement 9.5, accuracy of real operations 4.5.7, assignment 5.2, base type 3.3, basic operation 3.3.3, conversion 4.6, error situation 11, exception 11, exception handler 11.2, frame 11.2, numeric_error exception 11.1, predefined operator 4.5, predefined subprogram 8.6, propagation of an exception 11.4, real type 3.5.6, undefined value 3.2.1

11.7 Suppressing Checks

1   The presence of a SUPPRESS pragma gives permission to an implementation to omit certain run-time checks. The form of this pragma is as follows:

```
pragma SUPPRESS(identifier [, [ON =>] name]);
```

2   The identifier is that of the check that can be omitted. The name (if present) must be either a simple name or an expanded name and it must denote either an object, a type or subtype, a task unit, or a generic unit; alternatively the name can be a subprogram name, in which case it can stand for several visible overloaded subprograms.

3   A pragma SUPPRESS is only allowed immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated with the innermost enclosing block

statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

4   If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit.

In addition to the pragma SUPPRESS, DEC Ada provides the pragma SUPPRESS_ALL for the purpose of suppressing all run-time checks in a compilation unit. The form of this pragma is as follows:

```
pragma SUPPRESS_ALL;
```

The pragma SUPPRESS_ALL is only allowed following a compilation unit. The scope of the pragma is the entire unit or subunit that it follows.

5   The following checks correspond to situations in which the exception CONSTRAINT_ERROR may be raised; for these checks, the name (if present) must denote either an object or a type.

6   ACCESS_CHECK   When accessing a selected component, an indexed component, a slice, or an attribute, of an object designated by an access value, check that the access value is not null.

7   DISCRIMINANT_CHECK   Check that a discriminant of a composite value has the value imposed by a discriminant constraint. Also, when accessing a record component, check that it exists for the current discriminant values.

8   INDEX_CHECK   Check that the bounds of an array value are equal to the corresponding bounds of an index constraint. Also, when accessing a component of an array object, check for each dimension that the given index value belongs to the range defined by the bounds of the array object. Also, when accessing a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object.

9   LENGTH_CHECK   Check that there is a matching component
                                for each component of an array, in the case
                                of array assignments, type conversions,
                                and logical operators for arrays of boolean
                                components.

10   RANGE_CHECK   Check that a value satisfies a range
                                constraint. Also, for the elaboration of a
                                subtype indication, check that the constraint
                                (if present) is compatible with the type
                                mark. Also, for an aggregate, check that
                                an index or discriminant value belongs to
                                the corresponding subtype. Finally, check
                                for any constraint checks performed by a
                                generic instantiation.

                                In DEC Ada, when explicitly passing
                                an array by reference to an imported
                                subprogram, check if the array is aligned
                                on a byte boundary. Also, when a descriptor
                                or dope vector is created or used to pass
                                a parameter to or accept a function result
                                from an imported subprogram, check that
                                the descriptor or dope vector length field is
                                large enough to hold the actual parameter
                                or result length. See the DEC Ada run-time
                                reference manuals for more information on
                                when descriptors or dope vectors are created
                                or used.

11   The following checks correspond to situations in which the exception
      NUMERIC_ERROR is raised. The only allowed names in the corresponding
      pragmas are names of numeric types.

12   DIVISION_CHECK   Check that the second operand is not zero
                                for the operations /, **rem** and **mod.**

13   OVERFLOW_CHECK   Check that the result of a numeric operation
                                does not overflow.

14   The following check corresponds to situations in which the exception
      PROGRAM_ERROR is raised. The only allowed names in the corresponding
      pragmas are names denoting task units, generic units, or subprograms.

15   ELABORATION_CHECK   When either a subprogram is called, a task
                                     activation is accomplished, or a generic
                                     instantiation is elaborated, check that the
                                     body of the corresponding unit has already
                                     been elaborated.

16   The following check corresponds to situations in which the exception
     STORAGE_ERROR is raised. The only allowed names in the corresponding
     pragmas are names denoting access types, task units, or subprograms.

17   STORAGE_CHECK   Check that execution of an allocator does
                                     not require more space than is available for
                                     a collection. Check that the space available
                                     for a task or subprogram has not been
                                     exceeded.

18   If an error situation arises in the absence of the corresponding run-time
     checks, the execution of the program is erroneous (the results are not defined
     by the language).

19   **Examples:**

```
pragma SUPPRESS(RANGE_CHECK);
pragma SUPPRESS(INDEX_CHECK, ON => TABLE);
```

Notes:

20   For certain implementations, it may be impossible or too costly to suppress
     certain checks. The corresponding SUPPRESS pragma can be ignored. Hence,
     the occurrence of such a pragma within a given unit does not guarantee
     that the corresponding exception will not arise; the exceptions may also be
     propagated by called units.

     In DEC Ada, the pragma SUPPRESS_ALL does not suppress some checks that
     are always performed by the hardware and run-time system. For example, on
     OpenVMS systems, the checks DIVISION_CHECK and OVERFLOW_CHECK
     correspond to hardware checks that cannot be suppressed. The exceptions
     that correspond to the checks ACCESS_CHECK and STORAGE_CHECK
     may also be raised when the pragma SUPPRESS_ALL is in effect. For more
     information, see the DEC Ada run-time reference manuals.

21   **References:** access type 3.8, access value 3.8, activation 9.3, aggregate 4.3, allocator
     4.8, array 3.6, attribute 4.1.4, block statement 5.6, collection 3.8, compatible 3.3.2,
     component of an array 3.6, component of a record 3.7, composite type 3.3, constraint
     3.3, constraint_error exception 11.1, declarative part 3.9, designate 3.8, dimension 3.6,
     discrete range 3.6, discriminant 3.7.1, discriminant constraint 3.7.2, elaboration 3.1
     3.9, erroneous 1.6, error situation 11, expanded name 4.1.3, generic body 11.1, generic
     instantiation 12.3, generic unit 12, identifier 2.3, index 3.6, index constraint 3.6.1,
     indexed component 4.1.1, null access value 3.8, numeric operation 3.5.5 3.5.8 3.5.10,

numeric type 3.5, numeric_error exception 11.1, object 3.2, operation 3.3.3, package body 7.1, package specification 7.1, pragma 2.8, program_error exception 11.1, program unit 6, propagation of an exception 11.4, range constraint 3.5, record type 3.7, simple name 4.1, slice 4.1.2, subprogram 6, subprogram body 6.3, subprogram call 6.4, subtype 3.3, subunit 10.2, task 9, task body 9.1, task type 9.1, task unit 9, type 3.3, type mark 3.3.2

actual parameter 6.4 6.4.1, allow 1.6, compilation unit 10.1, function result 6.5, importing subprograms 13.9a.1.1, parameter 6.2, reference parameter passing mechanism 13.9a.1.4, scope 8.2

12

Generic Units

1   A generic unit is a program unit that is either a generic subprogram or a generic package. A generic unit is a *template*, which is parameterized or not, and from which corresponding (nongeneric) subprograms or packages can be obtained. The resulting program units are said to be *instances* of the original generic unit.

2   A generic unit is declared by a generic declaration. This form of declaration has a generic formal part declaring any generic formal parameters. An instance of a generic unit is obtained as the result of a generic instantiation with appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram. An instance of a generic package is a package.

3   Generic units are templates. As templates they do not have the properties that are specific to their nongeneric counterparts. For example, a generic subprogram can be instantiated but it cannot be called. In contrast, the instance of a generic subprogram is a nongeneric subprogram; hence, this instance can be called but it cannot be used to produce further instances.

4   **References:** declaration 3.1, generic actual parameter 12.3, generic declaration 12.1, generic formal parameter 12.1, generic formal part 12.1, generic instantiation 12.3, generic package 12.1, generic subprogram 12.1, instance 12.3, package 7, program unit 6, subprogram 6

12.1 Generic Declarations

1   A generic declaration declares a generic unit, which is either a generic subprogram or a generic package. A generic declaration includes a generic formal part declaring any generic formal parameters. A generic formal parameter can be an object; alternatively (unlike a parameter of a subprogram), it can be a type or a subprogram.

2   generic_declaration ::= generic_specification;

     generic_specification ::=
         generic_formal_part subprogram_specification
       | generic_formal_part package_specification

     generic_formal_part ::=
        generic {generic_parameter_declaration}

     generic_parameter_declaration ::=
         identifier_list : [**in** [**out**]] type_mark [:= expression];
       | **type** identifier **is** generic_type_definition;
       | private_type_declaration
       | **with** subprogram_specification [**is** name];
       | **with** subprogram_specification [**is** <>];

     generic_type_definition ::=
         (<>) | **range** <> | **digits** <> | **delta** <>
       | array_type_definition | access_type_definition

3   The terms generic formal object (or simply, *formal object*), generic formal type
    (or simply, *formal type*), and generic formal subprogram (or simply, *formal
    subprogram*) are used to refer to corresponding generic formal parameters.

4   The only form of subtype indication allowed within a generic formal part
    is a type mark (that is, the subtype indication must not include an explicit
    constraint). The designator of a generic subprogram must be an identifier.

5   Outside the specification and body of a generic unit, the name of this program
    unit denotes the generic unit. In contrast, within the declarative region
    associated with a generic subprogram, the name of this program unit denotes
    the subprogram obtained by the current instantiation of the generic unit.
    Similarly, within the declarative region associated with a generic package,
    the name of this program unit denotes the package obtained by the current
    instantiation.[1]

6   The elaboration of a generic declaration has no other effect.

7   **Examples of generic formal parts:**

```
generic      -- parameterless

generic
   SIZE : NATURAL;  --  formal object

generic
   LENGTH : INTEGER := 200;            -- formal object with a
                                       -- default expression

   AREA   : INTEGER := LENGTH*LENGTH; -- formal object with a
                                       -- default expression
```

---

[1] See also Appendix G, AI-00286, AI-00367, and AI-00412.

```
generic
   type ITEM  is private;                        -- formal type
   type INDEX is (<>);                            -- formal type
   type ROW   is array(INDEX range <>) of ITEM; -- formal type
   with function "<"(X, Y : ITEM) return BOOLEAN; -- formal
                                                  -- subprogram
```

8   **Examples of generic declarations declaring generic subprograms:**

```
generic
   type ELEM is private;
procedure EXCHANGE(U, V : in out ELEM);

generic
   type ITEM is private;
   with function "*"(U, V : ITEM) return ITEM is <>;
function SQUARING(X : ITEM) return ITEM;
```

9   **Example of a generic declaration declaring a generic package:**

```
generic
   type ITEM   is private;
   type VECTOR is array (POSITIVE range <>) of ITEM;
   with function SUM(X, Y : ITEM) return ITEM;
package ON_VECTORS is
   function SUM  (A, B : VECTOR) return VECTOR;
   function SIGMA(A    : VECTOR) return ITEM;
   LENGTH_ERROR : exception;
end;
```

Notes:

10   Within a generic subprogram, the name of this program unit acts as the name
     of a subprogram. Hence this name can be overloaded, and it can appear
     in a recursive call of the current instantiation. For the same reason, this
     name cannot appear after the reserved word **new** in a (recursive) generic
     instantiation.

11   An expression that occurs in a generic formal part is either the default
     expression for a generic formal object of mode **in**, or a constituent of an
     entry name given as default name for a formal subprogram, or the default
     expression for a parameter of a formal subprogram. Default expressions
     for generic formal objects and default names for formal subprograms are only
     evaluated for generic instantiations that use such defaults. Default expressions
     for parameters of formal subprograms are only evaluated for calls of the formal
     subprograms that use such defaults. (The usual visibility rules apply to any
     name used in a default expression: the denoted entity must therefore be visible
     at the place of the expression.)

12   Neither generic formal parameters nor their attributes are allowed constituents
     of static expressions (see 4.9).

₁₃ **References:** access type definition 3.8, array type definition 3.6, attribute 4.1.4, constraint 3.3, declaration 3.1, designator 6.1, elaboration has no other effect 3.1, entity 3.1, expression 4.4, function 6.5, generic instantiation 12.3, identifier 2.3, identifier list 3.2, instance 12.3, name 4.1, object 3.2, overloading 6.6 8.7, package specification 7.1, parameter of a subprogram 6.2, private type definition 7.4, procedure 6.1, reserved word 2.9, static expression 4.9, subprogram 6, subprogram specification 6.1, subtype indication 3.3.2, type 3.3, type mark 3.3.2

12.1.1 Generic Formal Objects

₁ The first form of generic parameter declaration declares generic formal objects. The type of a generic formal object is the base type of the type denoted by the type mark given in the generic parameter declaration. A generic parameter declaration with several identifiers is equivalent to a sequence of single generic parameter declarations, as explained in section 3.2.

₂ A generic formal object has a mode that is either **in** or **in out.** In the absence of an explicit mode indication in a generic parameter declaration, the mode **in** is assumed; otherwise the mode is the one indicated. If a generic parameter declaration ends with an expression, the expression is the *default expression* of the generic formal parameter. A default expression is only allowed if the mode is **in** (whether this mode is indicated explicitly or implicitly). The type of a default expression must be that of the corresponding generic formal parameter.

₃ A generic formal object of mode **in** is a constant whose value is a copy of the value supplied as the matching generic actual parameter in a generic instantiation, as described in section 12.3. The type of a generic formal object of mode **in** must not be a limited type; the subtype of such a generic formal object is the subtype denoted by the type mark given in the generic parameter declaration.

₄ A generic formal object of mode **in out** is a variable and denotes the object supplied as the matching generic actual parameter in a generic instantiation, as described in section 12.3. The constraints that apply to the generic formal object are those of the corresponding generic actual parameter.

Note:

₅ The constraints that apply to a generic formal object of mode **in out** are those of the corresponding generic actual parameter (not those implied by the type mark that appears in the generic parameter declaration). Whenever possible (to avoid confusion) it is recommended that the name of a base type be used for the declaration of such a formal object. If, however, the base type is anonymous, it is recommended that the subtype name defined by the type declaration for the base type be used.

6   **References:** anonymous type 3.3.1, assignment 5.2, base type 3.3, constant declaration 3.2, constraint 3.3, declaration 3.1, generic actual parameter 12.3, generic formal object 12.1, generic formal parameter 12.1, generic instantiation 12.3, generic parameter declaration 12.1, identifier 2.3, limited type 7.4.4, matching generic actual parameter 12.3, mode 6.1, name 4.1, object 3.2, simple name 4.1, subtype 3.3, type declaration 3.3, type mark 3.3.2, variable 3.2.1

12.1.2 Generic Formal Types

1   A generic parameter declaration that includes a generic type definition or a private type declaration declares a generic formal type. A generic formal type denotes the subtype supplied as the corresponding actual parameter in a generic instantiation, as described in 12.3(d). However, within a generic unit, a generic formal type is considered as being distinct from all other (formal or nonformal) types. The form of constraint applicable to a formal type in a subtype indication depends on the class of the type as for a nonformal type.

2   The only form of discrete range that is allowed within the declaration of a generic formal (constrained) array type is a type mark.

3   The discriminant part of a generic formal private type must not include a default expression for a discriminant. (Consequently, a variable that is declared by an object declaration must be constrained if its type is a generic formal type with discriminants.)

4   Within the declaration and body of a generic unit, the operations available for values of a generic formal type (apart from any additional operation specified by a generic formal subprogram) are determined by the generic parameter declaration for the formal type:

5   (a)   For a private type declaration, the available operations are those defined in section 7.4.2 (in particular, assignment, equality, and inequality are available for a private type unless it is limited).

6   (b)   For an array type definition, the available operations are those defined in section 3.6.2 (for example, they include the formation of indexed components and slices).

7   (c)   For an access type definition, the available operations are those defined in section 3.8.2 (for example, allocators can be used).

8   The four forms of generic type definition in which a *box* appears (that is, the compound delimiter <>) correspond to the following major forms of scalar type:

9   (d)   Discrete types: <>

             The available operations are the operations common to enumeration and integer types; these are defined in section 3.5.5.

10 (e) Integer types: **range** <>

The available operations are the operations of integer types defined in section 3.5.5.

11 (f) Floating point types: **digits** <>

The available operations are those defined in section 3.5.8.

12 (g) Fixed point types: **delta** <>

The available operations are those defined in section 3.5.10.

13 In all of the above cases (a) through (f), each operation implicitly associated with a formal type (that is, other than an operation specified by a formal subprogram) is implicitly declared at the place of the declaration of the formal type. The same holds for a formal fixed point type, except for the multiplying operators that deliver a result of the type *universal_fixed* (see 4.5.5), since these special operators are declared in the package STANDARD.

14 For an instantiation of the generic unit, each of these operations is the corresponding basic operation or predefined operator of the matching actual type. For an operator, this rule applies even if the operator has been redefined for the actual type or for some parent type of the actual type.

15 **Examples of generic formal types:**
```
type ITEM is private;
type BUFFER(LENGTH : NATURAL) is limited private;

type ENUM  is (<>);
type INT   is range <>;
type ANGLE is delta <>;
type MASS  is digits <>;

type TABLE is array (ENUM) of ITEM;
```

16 **Example of a generic formal part declaring a formal integer type:**
```
generic
   type RANK is range <>;
   FIRST  : RANK := RANK'FIRST;
   SECOND : RANK := FIRST + 1;  --  the operator "+" of
                                --  the type RANK
```

17 **References:** access type definition 3.8, allocator 4.8, array type definition 3.6, assignment 5.2, body of a generic unit 12.2, class of type 3.3, constraint 3.3, declaration 3.1, declaration of a generic unit 12.1, discrete range 3.6, discrete type 3.5, discriminant part 3.7.1, enumeration type 3.5.1, equality 4.5.2, fixed point type 3.5.9, floating point type 3.5.7, generic actual type 12.3, generic formal part 12.1, generic formal subprogram 12.1.3, generic formal type 12.1, generic parameter declaration 12.1, generic type definition 12.1, indexed component 4.1.1, inequality 4.5.2, instantiation 12.3, integer type 3.5.4, limited private type 7.4.4, matching generic

actual type 12.3.2 12.3.3 12.3.4 12.3.5, multiplying operator 4.5 4.5.5, operation 3.3, operator 4.5, parent type 3.4, private type definition 7.4, scalar type 3.5, slice 4.1.2, standard package 8.6 C, subtype indication 3.3.2, type mark 3.3.2, universal_fixed 3.5.9

12.1.3 Generic Formal Subprograms

1 A generic parameter declaration that includes a subprogram specification declares a generic formal subprogram.

2 Two alternative forms of defaults can be specified in the declaration of a generic formal subprogram. In these forms, the subprogram specification is followed by the reserved word **is** and either a box or the name of a subprogram or entry. The matching rules for these defaults are explained in section 12.3.6.

3 A generic formal subprogram denotes the subprogram, enumeration literal, or entry supplied as the corresponding generic actual parameter in a generic instantiation, as described in section 12.3(f).

4 **Examples of generic formal subprograms:**
```
with function INCREASE(X : INTEGER) return INTEGER;
with function SUM(X, Y : ITEM) return ITEM;

with function "+"(X, Y : ITEM) return ITEM is <>;
with function IMAGE(X : ENUM) return STRING is ENUM'IMAGE;

with procedure UPDATE is DEFAULT_UPDATE;
```

Notes:

5 The constraints that apply to a parameter of a formal subprogram are those of the corresponding parameter in the specification of the matching actual subprogram (not those implied by the corresponding type mark in the specification of the formal subprogram). A similar remark applies to the result of a function. Whenever possible (to avoid confusion), it is recommended that the name of a base type be used rather than the name of a subtype in any declaration of a formal subprogram. If, however, the base type is anonymous, it is recommended that the subtype name defined by the type declaration be used.

6 The type specified for a formal parameter of a generic formal subprogram can be any visible type, including a generic formal type of the same generic formal part.

7 **References:** anonymous type 3.3.1, base type 3.3, box delimiter 12.1.2, constraint 3.3, designator 6.1, generic actual parameter 12.3, generic formal function 12.1, generic formal subprogram 12.1, generic instantiation 12.3, generic parameter declaration 12.1, identifier 2.3, matching generic actual subprogram 12.3.6, operator symbol 6.1, parameter of a subprogram 6.2, renaming declaration 8.5, reserved word 2.9, scope 8.2, subprogram 6, subprogram specification 6.1, subtype 3.3.2, type 3.3, type mark 3.3.2

12.1a Pragma INLINE_GENERIC

DEC Ada provides the pragma INLINE_GENERIC to direct the compiler to attempt inline expansion of the body created for each instantiation of the named generic declarations or for the particular named instance. The form of this pragma is as follows:

```
pragma INLINE_GENERIC (name {, name});
```

Each name is either the name of a generic declaration or the name of an instance of a generic declaration. The pragma INLINE_GENERIC is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

If the pragma appears at the place of a declarative item, each name must denote a generic subprogram or package, or a (nongeneric) subprogram or package that is an instance of a generic subprogram or package, declared by an earlier declarative item of the same declarative part or package specification. If several (nongeneric, overloaded) subprograms satisfy this requirement, the pragma applies to all of them. If the pragma appears after a given library unit, the only name allowed is the name of that unit. If the name of a subprogram that is an instance of a generic subprogram is mentioned in the pragma, it indicates that only inline expansion of the instance itself is desired. It does not indicate that inline expansion of calls of the subprogram is desired.

If the name specified by a pragma INLINE_GENERIC is an instantiation declared by a renaming declaration, the pragma INLINE_GENERIC applies to the instantiation only if the instantiation that has been renamed, the renaming declaration, and the pragma all occur in the same declarative part or package specification. The pragma is ignored if these conditions are not satisfied.

The meaning of an instantiation is not changed by the pragma INLINE_ GENERIC.

Inline expansion of an instance creates a dependence of the unit containing the instantiation upon the corresponding generic proper body (the template and its subunits, if any). DEC Ada recognizes this dependence when deciding on the need for recompilation. See the DEC Ada program development manuals for more information on DEC Ada recompilation requirements.

Notes:

The pragma INLINE_GENERIC causes inline expansion of the generic body (and substitution of actual parameters for any generic formal parameters) at the point of any instantiation to which the pragma applies. Because of this effect, the pragma INLINE_GENERIC differs from the pragma INLINE in two respects. First, the pragma INLINE_GENERIC for a generic subprogram or an instance of a generic subprogram does not indicate a desire that calls of the subprograms are to be expanded inline. Second, the pragma INLINE_GENERIC can be given for a generic package or for an instance of a generic package (while the pragma INLINE cannot).

If the pragma INLINE is given for a generic subprogram, the pragma INLINE_GENERIC may also be given. Similarly, if the pragma INLINE is given for an instance of a generic subprogram, the pragma INLINE_GENERIC may also be given. In such cases, if calls are expanded inline, then the pragma INLINE_GENERIC has no additional effect. However, if calls are not expanded inline, the instance may still be expanded inline. (If only the pragma INLINE is given, and calls are not expanded inline, then the instance cannot be expanded inline either.)

If a pragma INLINE_GENERIC appears at the place of a declarative item and a name in the pragma is overloaded, the pragma applies only to those instantiations whose declarations occur (explicitly) earlier in the same declarative part or package specification.

References: compilation unit 10.1, declarative item 3.9, declarative part 3.9, generic declaration 12.1, generic package 12.1, generic subprogram 12.1, generic template 12 12.2, instance 12.3, instantiation 12.3, library unit 10.1, name 4.1, package specification 7.1, renaming declaration 8.5, subprogram 6, subunit 10.2

12.1b Pragma SHARE_GENERIC

DEC Ada provides the pragma SHARE_GENERIC to direct the compiler to attempt sharing of the code generated for each instantiation of the named generic declarations or for the particular named instances. The form of this pragma is as follows:

```
pragma SHARE_GENERIC (name {, name});
```

Each name is either the name of a generic declaration or the name of an instance of a generic declaration. The pragma SHARE_GENERIC is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

If the pragma appears at the place of a declarative item, each name must denote a generic subprogram or package, or a (nongeneric) subprogram or package that is an instance of a generic subprogram or package, declared by an earlier declarative item of the same declarative part or package specification. If several (nongeneric, overloaded) subprograms satisfy this requirement, the pragma applies to all of them. If the pragma appears after a given library unit, the only name allowed is the name of that unit.

If the name specified by a pragma SHARE_GENERIC is an instantiation declared by a renaming declaration, the pragma SHARE_GENERIC applies to the instantiation only if the instantiation that has been renamed, the renaming declaration, and the pragma all occur in the same declarative part or package specification. The pragma is ignored if these conditions are not satisfied.

The meaning of an instantiation is not changed by the pragma SHARE_GENERIC.

The pragmas SHARE_GENERIC and INLINE_GENERIC cannot apply to the same generic declaration. However, the pragma INLINE_GENERIC can be specified for an instance even if the pragma SHARE_GENERIC applies to the corresponding generic declaration. In this case, the pragma specified for the instance overrides the pragma that applies to the generic declaration, and the expansion of the particular instance is expanded inline. Similarly, the pragma SHARE_GENERIC can be specified for an instance even if the pragma INLINE_GENERIC applies to the corresponding generic declaration. Again, the pragma specified for the instance overrides the pragma that applies to the generic declaration.

Notes:

The pragma SHARE_GENERIC causes the code for an instance to be generated in such a manner as to allow the same code to be shared by other instances of the same generic under some conditions.

References: compilation unit 10.1, declarative item 3.9, declarative part 3.9, generic declaration 12.1, generic package 12.1, generic subprogram 12.1, generic template 12 12.2, instance 12.3, instantiation 12.3, library unit 10.1, name 4.1, package specification 7.1, renaming declaration 8.5, subprogram 6, subunit 10.2

12.2 Generic Bodies

1   The body of a generic subprogram or generic package is a template for the bodies of the corresponding subprograms or packages obtained by generic instantiations. The syntax of a generic body is identical to that of a nongeneric body.[2]

---

[2]   See also Appendix G, AI-00328.

2   For each declaration of a generic subprogram, there must be a corresponding body.

3   The elaboration of a generic body has no other effect than to establish that the body can from then on be used as the template for obtaining the corresponding instances.

4   **Example of a generic procedure body:**
```
procedure EXCHANGE(U, V : in out ELEM) is -- see example in 12.1
   T : ELEM;  --  the generic formal type
begin
   T := U;
   U := V;
   V := T;
end EXCHANGE;
```

5   **Example of a generic function body:**
```
function SQUARING(X : ITEM) return ITEM is -- see example in 12.1
begin
   return X*X;  --  the formal operator "*"
end;
```

6   **Example of a generic package body:**
```
package body ON_VECTORS is  --  see example in 12.1

   function SUM(A, B : VECTOR) return VECTOR is
      RESULT : VECTOR(A'RANGE);      --  the formal type VECTOR
      BIAS   : constant INTEGER := B'FIRST - A'FIRST;
   begin
      if A'LENGTH /= B'LENGTH then
         raise LENGTH_ERROR;
      end if;
      for N in A'RANGE loop
         RESULT(N) := SUM(A(N), B(N + BIAS));   --  the formal
                                                --  function SUM
      end loop;
      return RESULT;
   end;

   function SIGMA(A : VECTOR) return ITEM is
      TOTAL : ITEM := A(A'FIRST);                -- the formal
                                                 -- type ITEM
   begin
      for N in A'FIRST + 1 .. A'LAST loop
         TOTAL := SUM(TOTAL, A(N));              -- the formal
                                                 -- function SUM
      end loop;
      return TOTAL;
   end;
end;
```

7   **References:** body 3.9, elaboration 3.9, generic body 12.1, generic instantiation 12.3, generic package 12.1, generic subprogram 12.1, instance 12.3, package body 7.1, package 7, subprogram 6, subprogram body 6.3

12.3 **Generic Instantiation**

1   An instance of a generic unit is declared by a generic instantiation.

```
2       generic_instantiation ::=
            package identifier is
                new generic_package_name [generic_actual_part];
          | procedure identifier is
                new generic_procedure_name [generic_actual_part];
          | function designator is
                new generic_function_name [generic_actual_part];

        generic_actual_part ::=
           (generic_association {, generic_association})

        generic_association ::=
           [generic_formal_parameter =>] generic_actual_parameter

        generic_formal_parameter ::=
           parameter_simple_name | operator_symbol

        generic_actual_parameter ::= expression | variable_name
             | subprogram_name | entry_name | type_mark
```

3   An explicit generic actual parameter must be supplied for each generic formal parameter, unless the corresponding generic parameter declaration specifies that a default can be used. Generic associations can be either positional or named in the same manner as parameter associations of subprogram calls (see 6.4). If two or more formal subprograms have the same designator, then named associations are not allowed for the corresponding generic parameters.

4   Each generic actual parameter must *match* the corresponding generic formal parameter. An expression can match a formal object of mode **in**; a variable name can match a formal object of mode **in out**; a subprogram name or an entry name can match a formal subprogram; a type mark can match a formal type. The detailed rules defining the allowed matches are given in sections 12.3.1 to 12.3.6; these are the only allowed matches.

5   The instance is a copy of the generic unit, apart from the generic formal part; thus the instance of a generic package is a package, that of a generic procedure is a procedure, and that of a generic function is a function. For each occurrence, within the generic unit, of a name that denotes a given entity, the following list defines which entity is denoted by the corresponding occurrence within the instance. [3]

---

[3]   See also Appendix G, AI-00398, AI-00409, and AI-00483.

6   (a)   For a name that denotes the generic unit: The corresponding occurrence denotes the instance.

7   (b)   For a name that denotes a generic formal object of mode **in**: The corresponding name denotes a constant whose value is a copy of the value of the associated generic actual parameter.

8   (c)   For a name that denotes a generic formal object of mode **in out**: The corresponding name denotes the variable named by the associated generic actual parameter.

9   (d)   For a name that denotes a generic formal type: The corresponding name denotes the subtype named by the associated generic actual parameter (the actual subtype).

10   (e)   For a name that denotes a discriminant of a generic formal type: The corresponding name denotes the corresponding discriminant (there must be one) of the actual type associated with the generic formal type.

11   (f)   For a name that denotes a generic formal subprogram: The corresponding name denotes the subprogram, enumeration literal, or entry named by the associated generic actual parameter (the actual subprogram).

12   (g)   For a name that denotes a formal parameter of a generic formal subprogram: The corresponding name denotes the corresponding formal parameter of the actual subprogram associated with the formal subprogram.

13   (h)   For a name that denotes a local entity declared within the generic unit: The corresponding name denotes the entity declared by the corresponding local declaration within the instance.

14   (i)   For a name that denotes a global entity declared outside of the generic unit: The corresponding name denotes the same global entity.

15   Similar rules apply to operators and basic operations: in particular, formal operators follow a rule similar to rule (f), local operations follow a rule similar to rule (h), and operations for global types follow a rule similar to rule (i). In addition, if within the generic unit a predefined operator or basic operation of a formal type is used, then within the instance the corresponding occurrence refers to the corresponding predefined operation of the actual type associated with the formal type.

16   The above rules apply also to any type mark or (default) expression given within the generic formal part of the generic unit.

17 For the elaboration of a generic instantiation, each expression supplied as an explicit generic actual parameter is first evaluated, as well as each expression that appears as a constituent of a variable name or entry name supplied as an explicit generic actual parameter; these evaluations proceed in some order that is not defined by the language. Then, for each omitted generic association (if any), the corresponding default expression or default name is evaluated; such evaluations are performed in the order of the generic parameter declarations. Finally, the implicitly generated instance is elaborated. The elaboration of a generic instantiation may also involve certain constraint checks as described in later subsections.[4]

18 Recursive generic instantiation is not allowed in the following sense: if a given generic unit includes an instantiation of a second generic unit, then the instance generated by this instantiation must not include an instance of the first generic unit (whether this instance is generated directly, or indirectly by intermediate instantiations).

19 **Examples of generic instantiations (see 12.1):**
```
procedure SWAP is new EXCHANGE(ELEM => INTEGER);
procedure SWAP is new EXCHANGE(CHARACTER); --  SWAP is overloaded

function SQUARE is new SQUARING(INTEGER);  --  "*" of INTEGER
                                           --  used by default

function SQUARE is new SQUARING(ITEM => MATRIX,
                               "*"  => MATRIX_PRODUCT);

function SQUARE is new SQUARING(MATRIX, MATRIX_PRODUCT);
                                          -- same as previous

package INT_VECTORS is new ON_VECTORS(INTEGER, TABLE, "+");
```

20 **Examples of uses of instantiated units:**
```
SWAP(A, B);
A := SQUARE(A);

T : TABLE(1 .. 5) := (10, 20, 30, 40, 50);
N : INTEGER := INT_VECTORS.SIGMA(T);  --  150 (see 12.2 for
                                      --  the body of SIGMA)

use INT_VECTORS;
M : INTEGER := SIGMA(T);  --  150
```

---

[4] See also Appendix G, AI-00237 and AI-00365.

Notes:

21 Omission of a generic actual parameter is only allowed if a corresponding default exists. If default expressions or default names (other than simple names) are used, they are evaluated in the order in which the corresponding generic formal parameters are declared.

22 If two overloaded subprograms declared in a generic package specification differ only by the (formal) type of their parameters and results, then there exist legal instantiations for which all calls of these subprograms from outside the instance are ambiguous. For example:

```
generic
   type A is (<>);
   type B is private;
package G is
   function NEXT(X : A) return A;
   function NEXT(X : B) return B;
end;

package P is new G(A => BOOLEAN, B => BOOLEAN);
-- calls of P.NEXT are ambiguous
```

23 **References:** declaration 3.1, designator 6.1, discriminant 3.7.1, elaboration 3.1 3.9, entity 3.1, entry name 9.5, evaluation 4.5, expression 4.4, generic formal object 12.1, generic formal parameter 12.1, generic formal subprogram 12.1, generic formal type 12.1, generic parameter declaration 12.1, global declaration 8.1, identifier 2.3, implicit declaration 3.1, local declaration 8.1, mode in 12.1.1, mode in out 12.1.1, name 4.1, operation 3.3, operator symbol 6.1, overloading 6.6 8.7, package 7, simple name 4.1, subprogram 6, subprogram call 6.4, subprogram name 6.1, subtype declaration 3.3.2, type mark 3.3.2, variable 3.2.1, visibility 8.3

12.3.1 Matching Rules for Formal Objects

1 A generic formal parameter of mode **in** of a given type is matched by an expression of the same type. If a generic unit has a generic formal object of mode **in**, a check is made that the value of the expression belongs to the subtype denoted by the type mark, as for an explicit constant declaration (see 3.2.1). The exception CONSTRAINT_ERROR is raised if this check fails.

2 A generic formal parameter of mode **in out** of a given type is matched by the name of a variable of the same type. The variable must not be a formal parameter of mode **out** or a subcomponent thereof. The name must denote a variable for which renaming is allowed (see 8.5).

Notes:

3   The type of a generic actual parameter of mode **in** must not be a limited type. The constraints that apply to a generic formal parameter of mode **in out** are those of the corresponding generic actual parameter (see 12.1.1).

4   **References:** constraint 3.3, constraint_error exception 11.1, expression 4.4, formal parameter 6.1, generic actual parameter 12.3, generic formal object 12.1.1, generic formal parameter 12.1, generic instantiation 12.3, generic unit 12.1, limited type 7.4.4, matching generic actual parameter 12.3, mode in 12.1.1, mode in out 12.1.1, mode out 6.2, name 4.1, raising of exceptions 11, satisfy 3.3, subcomponent 3.3, type 3.3, type mark 3.3.2, variable 3.2.1

12.3.2 Matching Rules for Formal Private Types

1   A generic formal private type is matched by any type or subtype (the actual subtype) that satisfies the following conditions:

2   •   If the formal type is not limited, the actual type must not be a limited type. (If, on the other hand, the formal type is limited, no such condition is imposed on the corresponding actual type, which can be limited or not limited.)

3   •   If the formal type has a discriminant part, the actual type must be a type with the same number of discriminants; the type of a discriminant that appears at a given position in the discriminant part of the actual type must be the same as the type of the discriminant that appears at the same position in the discriminant part of the formal type; and the actual subtype must be unconstrained. (If, on the other hand, the formal type has no discriminants, the actual type is allowed to have discriminants.)

4   Furthermore, consider any occurrence of the name of the formal type at a place where this name is used as an unconstrained subtype indication. The actual subtype must not be an unconstrained array type or an unconstrained type with discriminants, if any of these occurrences is at a place where either a constraint or default discriminants would be required for an array type or for a type with discriminants (see 3.6.1 and 3.7.2). The same restriction applies to occurrences of the name of a subtype of the formal type, and to occurrences of the name of any type or subtype derived, directly or indirectly, from the formal type.[5]

---

[5]   See also Appendix G, AI-00037.

5   If a generic unit has a formal private type with discriminants, the elaboration of a corresponding generic instantiation checks that the subtype of each discriminant of the actual type is the same as the subtype of the corresponding discriminant of the formal type. The exception CONSTRAINT_ERROR is raised if this check fails.

6   **References:** array type 3.6, constraint 3.3, constraint_error exception 11.1, default expression for a discriminant 3.7.1, derived type 3.4, discriminant 3.7.1, discriminant part 3.7.1, elaboration 3.9, generic actual type 12.3, generic body 12.2, generic formal type 12.1.2, generic instantiation 12.3, generic specification 12.1, limited type 7.4.4, matching generic actual parameter 12.3, name 4.1, private type 7.4, raising of exceptions 11, subtype 3.3, subtype indication 3.3.2, type 3.3, type with discriminants 3.3, unconstrained array type 3.6, unconstrained subtype 3.3

12.3.3 Matching Rules for Formal Scalar Types

1   A generic formal type defined by (<>) is matched by any discrete subtype (that is, any enumeration or integer subtype). A generic formal type defined by **range** <> is matched by any integer subtype. A generic formal type defined by **digits** <> is matched by any floating point subtype. A generic formal type defined by **delta** <> is matched by any fixed point subtype. No other matches are possible for these generic formal types.

2   **References:** box delimiter 12.1.2, discrete type 3.5, enumeration type 3.5.1, fixed point type 3.5.9, floating point type 3.5.7, generic actual type 12.3, generic formal type 12.1.2, generic type definition 12.1, integer type 3.5.4, matching generic actual parameter 12.3, scalar type 3.5

12.3.4 Matching Rules for Formal Array Types

1   A formal array type is matched by an actual array subtype that satisfies the following conditions:

2   • The formal array type and the actual array type must have the same dimensionality; the formal type and the actual subtype must be either both constrained or both unconstrained.

3   • For each index position, the index type must be the same for the actual array type as for the formal array type.

4   • The component type must be the same for the actual array type as for the formal array type. If the component type is other than a scalar type, then the component subtypes must be either both constrained or both unconstrained.

5   If a generic unit has a formal array type, the elaboration of a corresponding
    instantiation checks that the constraints (if any) on the component type are the
    same for the actual array type as for the formal array type, and likewise that
    for any given index position the index subtypes or the discrete ranges have
    the same bounds. The exception CONSTRAINT_ERROR is raised if this check
    fails.

6   **Example:**

```
-- given the generic package

generic
   type ITEM   is private;
   type INDEX  is (<>);
   type VECTOR is array (INDEX range <>) of ITEM;
   type TABLE  is array (INDEX) of ITEM;
package P is
   ...
end;

-- and the types

type MIX    is array (COLOR range <>) of BOOLEAN;
type OPTION is array (COLOR) of BOOLEAN;

-- then MIX can match VECTOR and OPTION can match TABLE

package R is new P(ITEM   => BOOLEAN, INDEX => COLOR,
                   VECTOR => MIX,     TABLE => OPTION);

-- Note that MIX cannot match TABLE and
-- OPTION cannot match VECTOR
```

Note:

7   For the above rules, if any of the index or component types of the formal array
    type is itself a formal type, then within the instance its name denotes the
    corresponding actual subtype (see 12.3(d)).

8   **References:** array type 3.6, array type definition 3.6, component of an array 3.6,
    constrained array type 3.6, constraint 3.3, constraint_error exception 11.1, elaboration
    3.9, formal type 12.1, generic formal type 12.1.2, generic instantiation 12.3, index 3.6,
    index constraint 3.6.1, matching generic actual parameter 12.3, raise statement 11.3,
    subtype 3.3, unconstrained array type 3.6

12.3.5 Matching Rules for Formal Access Types

1   A formal access type is matched by an actual access subtype if the type of the
    designated objects is the same for the actual type as for the formal type. If the
    designated type is other than a scalar type, then the designated subtypes must
    be either both constrained or both unconstrained.

2   If a generic unit has a formal access type, the elaboration of a corresponding instantiation checks that any constraints on the designated objects are the same for the actual access subtype as for the formal access type. The exception CONSTRAINT_ERROR is raised if this check fails.

3   **Example:**

```
-- the formal types of the generic package

generic
   type NODE is private;
   type LINK is access NODE;
package P is
   ...
end;

-- can be matched by the actual types

type CAR;
type CAR_NAME is access CAR;

type CAR is
   record
      PRED, SUCC : CAR_NAME;
      NUMBER     : LICENSE_NUMBER;
      OWNER      : PERSON;
   end record;

-- in the following generic instantiation

package R is new P(NODE => CAR, LINK => CAR_NAME);
```

Note:

4   For the above rules, if the designated type is itself a formal type, then within the instance its name denotes the corresponding actual subtype (see 12.3(d)).

5   **References:** access type 3.8, access type definition 3.8, constraint 3.3, constraint_ error exception 11.1, designate 3.8, elaboration 3.9, generic formal type 12.1.2, generic instantiation 12.3, matching generic actual parameter 12.3, object 3.2, raise statement 11.3, value of access type 3.8

12.3.6 Matching Rules for Formal Subprograms

1   A formal subprogram is matched by an actual subprogram, enumeration literal, or entry if both have the same parameter and result type profile (see 6.6); in addition, parameter modes must be identical for formal parameters that are at the same parameter position.

2   If a generic unit has a default subprogram specified by a name, this name must denote a subprogram, an enumeration literal, or an entry, that matches the formal subprogram (in the above sense). The evaluation of the default name

takes place during the elaboration of each instantiation that uses the default, as defined in section 12.3.[6]

3   If a generic unit has a default subprogram specified by a box, the corresponding actual parameter can be omitted if a subprogram, enumeration literal, or entry matching the formal subprogram, and with the same designator as the formal subprogram, is directly visible at the place of the generic instantiation; this subprogram, enumeration literal, or entry is then used by default (there must be exactly one subprogram, enumeration literal, or entry satisfying the previous conditions).

4   **Example:**

```
-- given the generic function specification

generic
   type ITEM is private;
   with function "*" (U, V : ITEM) return ITEM is <>;
function SQUARING(X : ITEM) return ITEM;

-- and the function

function MATRIX_PRODUCT(A, B : MATRIX) return MATRIX;

-- the following instantiation is possible

function SQUARE is new SQUARING(MATRIX, MATRIX_PRODUCT);

-- the following instantiations are equivalent

function SQUARE is new SQUARING(ITEM => INTEGER, "*" => "*");
function SQUARE is new SQUARING(INTEGER, "*");
function SQUARE is new SQUARING(INTEGER);
```

Notes:

5   The matching rules for formal subprograms state requirements that are similar to those applying to subprogram renaming declarations (see 8.5). In particular, the name of a parameter of the formal subprogram need not be the same as that of the corresponding parameter of the actual subprogram; similarly, for these parameters, default expressions need not correspond.

6   A formal subprogram is matched by an attribute of a type if the attribute is a function with a matching specification. An enumeration literal of a given type matches a parameterless formal function whose result type is the given type.

7   **References:** attribute 4.1.4, box delimiter 12.1.2, designator 6.1, entry 9.5, function 6.5, generic actual type 12.3, generic formal subprogram 12.1.3, generic formal type 12.1.2, generic instantiation 12.3, matching generic actual parameter 12.3, name 4.1, parameter and result type profile 6.3, subprogram 6, subprogram specification 6.1, subtype 3.3, visibility 8.3

---

[6] See also Appendix G, AI-00038.

12.4 Example of a Generic Package

1   The following example provides a possible formulation of stacks by means of a
     generic package. The size of each stack and the type of the stack elements are
     provided as generic parameters.

2
```
generic
   SIZE : POSITIVE;
   type ITEM is private;
package STACK is
   procedure PUSH(E : in  ITEM);
   procedure POP (E : out ITEM);
   OVERFLOW, UNDERFLOW : exception;
end STACK;

package body STACK is

   type TABLE is array (POSITIVE range <>) of ITEM;
   SPACE : TABLE(1 .. SIZE);
   INDEX : NATURAL := 0;

   procedure PUSH(E : in ITEM) is
   begin
      if INDEX >= SIZE then
         raise OVERFLOW;
      end if;
      INDEX := INDEX + 1;
      SPACE(INDEX) := E;
   end PUSH;

   procedure POP(E : out ITEM) is
   begin
      if INDEX = 0 then
         raise UNDERFLOW;
      end if;
      E := SPACE(INDEX);
      INDEX := INDEX - 1;
   end POP;

end STACK;
```

3   Instances of this generic package can be obtained as follows:

```
package STACK_INT  is new STACK(SIZE => 200, ITEM => INTEGER);
package STACK_BOOL is new STACK(100, BOOLEAN);
```

4   Thereafter, the procedures of the instantiated packages can be called as
     follows:

```
STACK_INT.PUSH(N);
STACK_BOOL.PUSH(TRUE);
```

5   Alternatively, a generic formulation of the type STACK can be given as follows
     (package body omitted):

```
generic
   type ITEM is private;
package ON_STACKS is
   type STACK(SIZE : POSITIVE) is limited private;
   procedure PUSH(S : in out STACK; E : in  ITEM);
   procedure POP (S : in out STACK; E : out ITEM);
   OVERFLOW, UNDERFLOW : exception;
private
   type TABLE is array (POSITIVE range <>) of ITEM;
   type STACK(SIZE : POSITIVE) is
      record
         SPACE : TABLE(1 .. SIZE);
         INDEX : NATURAL := 0;
      end record;
end;
```

6   In order to use such a package, an instantiation must be created and thereafter
     stacks of the corresponding type can be declared:

```
declare
   package STACK_REAL is new ON_STACKS(REAL); use STACK_REAL;
   S : STACK(100);
begin
   ...
   PUSH(S, 2.54);
   ...
end;
```

13

Representation Clauses and Implementation-Dependent Features

1   This chapter describes representation clauses, certain implementation-dependent features, and other features that are used in system programming.

13.1 Representation Clauses

1   Representation clauses specify how the types of the language are to be mapped onto the underlying machine. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).

2
```
representation_clause ::=
      type_representation_clause | address_clause

type_representation_clause ::= length_clause
    | enumeration_representation_clause
    | record_representation_clause
```

3   A type representation clause applies either to a type or to a *first named subtype* (that is, to a subtype declared by a type declaration, the base type being therefore anonymous). Such a representation clause applies to all objects that have this type or this first named subtype. At most one enumeration or record representation clause is allowed for a given type: an enumeration representation clause is only allowed for an enumeration type; a record representation clause, only for a record type. (On the other hand, more than one length clause can be provided for a given type; moreover, both a length clause and an enumeration or record representation clause can be provided.) A length clause is the only form of representation clause allowed for a type derived from a parent type that has (user-defined) derivable subprograms. [1]

4   An address clause applies either to an object; to a subprogram, package, or task unit; or to an entry. At most one address clause is allowed for any of these entities.

---

[1]   See also Appendix G, AI-00040, AI-00138, and AI-00422.

In DEC Ada, an address clause can apply to an object or imported subprogram. On Digital UNIX systems, an address clause can also apply to a single entry. See section 13.5 for more information.

5   A representation clause and the declaration of the entity to which the clause applies must both occur immediately within the same declarative part, package specification, or task specification; the declaration must occur before the clause. In the absence of a representation clause for a given declaration, a default representation of this declaration is determined by the implementation. Such a default determination occurs no later than the end of the immediately enclosing declarative part, package specification, or task specification. For a declaration given in a declarative part, this default determination occurs before any enclosed body.

6   In the case of a type, certain occurrences of its name imply that the representation of the type must already have been determined. Consequently these occurrences force the default determination of any aspect of the representation not already determined by a prior type representation clause. This default determination is also forced by similar occurrences of the name of a subtype of the type, or of the name of any type or subtype that has subcomponents of the type. A forcing occurrence is any occurrence other than in a type or subtype declaration, a subprogram specification, an entry declaration, a deferred constant declaration, a pragma, or a representation clause for the type itself. In any case, an occurrence within an expression is always forcing. [2]

7   A representation clause for a given entity must not appear after an occurrence of the name of the entity if this occurrence forces a default determination of representation for the entity.[3]

8   Similar restrictions exist for address clauses. For an object, any occurrence of its name (after the object declaration) is a forcing occurrence. For a subprogram, package, task unit, or entry, any occurrence of a representation attribute of such an entity is a forcing occurrence.

9   The effect of the elaboration of a representation clause is to define the corresponding aspects of the representation.

10   The interpretation of some of the expressions that appear in representation clauses is implementation-dependent, for example, expressions specifying addresses. An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware. If a representation clause is accepted by an implementation, the compiler

---

[2]   See also Appendix G, AI-00039, AI-00186, AI-00321, and AI-00322.
[3]   See also Appendix G, AI-00039 and AI-00371.

must guarantee that the net effect of the program is not changed by the presence of the clause, except for address clauses and for parts of the program that interrogate representation attributes. If a program contains a representation clause that is not accepted, the program is illegal. For each implementation, the allowed representation clauses, and the conventions used for implementation-dependent expressions, must be documented in Appendix F of the reference manual.

The allowed representation clauses and the conventions used for implementation-dependent expressions are documented in this chapter and summarized in Appendix F.

11 Whereas a representation clause is used to impose certain characteristics of the mapping of an entity onto the underlying machine, pragmas can be used to provide an implementation with criteria for its selection of such a mapping. The pragma PACK specifies that storage minimization should be the main criterion when selecting the representation of a record or array type. Its form is as follows:

```
pragma PACK(type_simple_name);
```

12 Packing means that gaps between the storage areas allocated to consecutive components should be minimized. It need not, however, affect the mapping of each component onto storage. This mapping can itself be influenced by a pragma (or controlled by a representation clause) for the component or component type. The position of a PACK pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause; in particular, the pragma must appear before any use of a representation attribute of the packed entity.

13 The pragma PACK is the only language-defined representation pragma. Additional representation pragmas may be provided by an implementation; these must be documented in Appendix F. (In contrast to representation clauses, a pragma that is not accepted by the implementation is ignored.)

In DEC Ada, each noncomposite component (scalar, access, and so on) is aligned by default on an appropriate boundary, according to the following conventions:

- On OpenVMS VAX systems, all noncomposite components are aligned on byte boundaries.

- On Alpha systems, all noncomposite components are aligned on natural boundaries. For example, 1-byte components are aligned on byte boundaries, 2-byte components on 2-byte boundaries, 4-byte components on 4-byte boundaries, and so on.

Each composite component is aligned so that it satisfies the maximum alignment required for its subcomponents and (in the case of record components) any representation clauses or representation pragmas that might apply.

On all systems, the effect of the pragma PACK on a record or array is to cause the record or array components to be allocated in the next available bit (if the component is packable) or byte (otherwise) without regard to natural boundaries. Whether any particular component is packable depends on the rules for its type. See the DEC Ada run-time reference manuals for more information.

DEC Ada provides one additional representation pragma: the pragma COMPONENT_ALIGNMENT. See section 13.1a for more information.

Note:

14 No representation clause is allowed for a generic formal type.

DEC Ada also does not allow a representation clause for a type that depends on a generic formal type. A type depends on a generic formal type if:

- It has a subcomponent of a generic formal type or a subcomponent that depends on a generic formal type

- If it is derived from a generic formal type or a type that depends on a generic formal type.

15 **References:** address clause 13.5, allow 1.6, body 3.9, component 3.3, declaration 3.1, declarative part 3.9, default expression 3.2.1, deferred constant declaration 7.4, derivable subprogram 3.4, derived type 3.4, entity 3.1, entry 9.5, enumeration representation clause 13.3, expression 4.4, generic formal type 12.1.2, illegal 1.6, length clause 13.2, must 1.6, name 4.1, object 3.2, occur immediately within 8.1, package 7, package specification 7.1, parent type 3.4, pragma 2.8, record representation clause 13.4, representation attribute 13.7.2 13.7.3, subcomponent 3.3, subprogram 6, subtype 3.3, subtype declaration 3.3.2, task specification 9.1, task unit 9, type 3.3, type declaration 3.3.1 extension>

array 3.6, constant 3.2.1, record 3.7, variable 3.2.1, variant 3.7.3

13.1a The Pragma Component_Alignment

DEC Ada provides the representation pragma COMPONENT_ALIGNMENT to allow the specification of a default storage representation for declarative parts or array or record types. The form of this pragma is as follows:

```
pragma COMPONENT_ALIGNMENT([FORM =>] alignment_choice
    [, [NAME =>] type_simple_name]);

alignment_choice ::=
    COMPONENT_SIZE | COMPONENT_SIZE_4
    | DEFAULT      | STORAGE_UNIT
```

The alignment choices are briefly defined as follows:

COMPONENT_SIZE
Specifies that scalar components and subcomponents of the array or record type are aligned on boundaries appropriate to their inherent size (naturally aligned). For example, 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.

COMPONENT_SIZE_4
Specifies that components with a size of four or fewer bytes are naturally aligned. Components that are larger than 4 bytes are placed on the next 4-byte boundary.

DEFAULT
Specifies that array or record components are aligned on default boundaries appropriate to the underlying hardware or operating system or both.

On OpenVMS VAX systems, the effect of the DEFAULT choice is the same as the STORAGE_UNIT choice (byte alignment).

On Alpha systems, the effect of the DEFAULT choice is the same as the COMPONENT_SIZE choice (natural alignment).

STORAGE_UNIT
Specifies that array or record components are aligned on boundaries determined by the current value of the constant SYSTEM.STORAGE_UNIT.

In DEC Ada, the value of the constant SYSTEM.STORAGE_UNIT must be 8 (bits) (1 byte). The STORAGE_UNIT alignment choice causes all array or record components to which the pragma COMPONENT_ALIGNMENT applies to be aligned on byte boundaries.

When you omit the NAME parameter, the pragma must occur within a declarative part or package specification. In this case, the pragma specifies a default representation for record or array types that are not specified in a representation clause or pragma. The effect of the pragma extends to types declared from the place of the pragma to the end of the innermost declarative part or package specification in which the pragma was declared.

When you specify the NAME parameter, the type simple name must be the name of an array or record type, and the pragma applies only to that type. The pragma and the type declaration must both occur immediately within the same declarative part, package specification, or task specification The declaration must occur before the pragma. The position of the pragma and the restrictions on the named type are governed by the same rules as those for a representation clause. In particular, the pragma must appear before any use of a representation attribute of the named type.

When the pragma COMPONENT_ALIGNMENT applies to a type, the following rules or consequences apply:

- A pragma PACK and a pragma COMPONENT_ALIGNMENT cannot both apply to a type. If both apply, then the pragma COMPONENT_ALIGNMENT is ignored.

- If a representation clause and a pragma COMPONENT_ALIGNMENT both apply to a type, the components mentioned in the representation clause are laid out first, and any remaining components are laid out and aligned in a manner consistent with the pragma.

If a pragma COMPONENT_ALIGNMENT does not apply to a type, then a default alignment is used (see 13.1). On OpenVMS VAX systems, the default alignment is byte alignment (equivalent to the STORAGE_UNIT alignment choice). On Alpha systems, the default alignment is natural alignment (equivalent to the COMPONENT_SIZE alignment choice).

Example:

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

In this example, the COMPONENT_SIZE choice causes the pragma COMPONENT_ALIGNMENT to align the component FLOAT_REC.SINGLE at byte 0 and the component FLOAT_REC.DOUBLE at byte 8. The STORAGE_UNIT choice would have aligned FLOAT_REC.DOUBLE at byte 4.

By explicitly specifying the COMPONENT_SIZE choice, the use of this pragma ensures that natural alignment is used across all OpenVMS and Digital UNIX systems.

Note:

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. For example, if a derived type inherits a pragma PACK, an explicit pragma COMPONENT_ALIGNMENT applied to that type overrides the inherited pragma PACK. Similarly, the alignment choice of a pragma COMPONENT_ALIGNMENT that applies to a derived type overrides the alignment choice of a pragma COMPONENT_ALIGNMENT that applies to the parent type.

References: array 3.6, array component 3.6, array type 3.6, compilation unit 10.1, component 3.3, declarative part 3.9, name 4.1, package specification 7.1, pragma 2.8, record 3.7, record component 3.7, record type 3.7, representation clause 13.1, task specification 9.1, type 3.3, type declaration 3.3

13.2 Length Clauses

1   A length clause specifies an amount of storage associated with a type.

2   ```
     length_clause ::= for attribute use simple_expression;
          4
     ```

3    The expression must be of some numeric type and is evaluated during the elaboration of the length clause (unless it is a static expression). The prefix of the attribute must denote either a type or a first named subtype. The prefix is called T in what follows. The only allowed attribute designators in a length

---

4  See also Appendix G, AI-00300.

clause are SIZE, STORAGE_SIZE, and SMALL. The effect of the length clause depends on the attribute designator:

4    (a)   Size specification: T'SIZE

5          The expression must be a static expression of some integer type. The value of the expression specifies an upper bound for the number of bits to be allocated to objects of the type or first named subtype T. The size specification must allow for enough storage space to accommodate every allowable value of these objects. A size specification for a composite type may affect the size of the gaps between the storage areas allocated to consecutive components. On the other hand, it need not affect the size of the storage area allocated to each component.

6          The size specification is only allowed if the constraints on T and on its subcomponents (if any) are static. In the case of an unconstrained array type, the index subtypes must also be static.

          In DEC Ada, for a discrete type, the given size must not exceed the maximum size (in bits) for the given type. On OpenVMS VAX systems, the maximum size is 32 for all types. On Alpha systems, the maximum size is 64 for integer and enumeration types and 32 for fixed point types. The given size becomes the default allocation for all objects and components (in arrays and records) of that type.

          For integer and enumeration types, the given size affects the internal representation as follows: for integer types, high order bits are sign-extended; for enumeration types, the high order bits can be either zero- or sign-extended depending upon the base representation that is selected.

          For fixed point types, the given size affects the range (but not the precision) of the underlying model numbers of the type. That is, the given size determines the value of B, which is described in section 3.5.9. The given size may not equal the value of B because the given size includes any sign bit and B does not.

          For all other types, the given size must equal the size that would apply in the absence of a size specification.

7    (b)   Specification of collection size: T'STORAGE_SIZE

8          The prefix T must denote an access type. The expression must be of some integer type (but need not be static); its value specifies the number of storage units to be reserved for the collection, that is, the storage space needed to contain all objects designated by values of the access type and by values of other types derived from the access type,

directly or indirectly. This form of length clause is not allowed for a type derived from an access type.

In DEC Ada, the specification of a collection size is interpreted as follows. If the value of the expression is greater than zero, the specified size (representing the number of bytes in the collection) is rounded up to an appropriate boundary and then is used as the initial size for the collection. The collection is not extended if the initial allocation is exhausted. The exception CONSTRAINT_ERROR is raised if the value, rounded up, exceeds the maximum collection size.

If the value is equal to zero or less than zero, no storage is allocated for the collection, and the collection is not extended.

The default behavior (in the absence of a length clause) is for no storage to be initially allocated for the collection. Storage is then allocated as needed, until all virtual memory is depleted.

9     (c)     Specification of storage for a task activation: T'STORAGE_SIZE

10              The prefix T must denote a task type. The expression must be of some integer type (but need not be static); its value specifies the number of storage units to be reserved for an activation (not the code) of a task of the type.

In DEC Ada, the specification of storage for a task activation is interpreted as follows. If the value of the expression is greater than zero, the specified storage (in bytes) is rounded up to an appropriate boundary and is then used as the amount of storage to be allocated for an activation of a task of the given type. If the value is equal to zero, a default allocation is used. (This is the default behavior in the absence of a length clause.)

In both cases, the task activation storage is fixed and is not extended if the initial allocation is exhausted. If the value is less than zero, the exception CONSTRAINT_ERROR is raised.

The storage allocation for a task may also be affected by the pragmas TASK_STORAGE (see 13.2a) and MAIN_STORAGE (see 13.2b); see also the DEC Ada run-time reference manuals.

11     (d)     Specification of *small* for a fixed point type: T'SMALL

12              The prefix T must denote the first named subtype of a fixed point type. The expression must be a static expression of some real type; its value must not be greater than the delta of the first named subtype. The effect of the length clause is to use this value of *small* for the representation of values of the fixed point base type. (The length

clause thereby also affects the amount of storage for objects that have this type.) [5]

In DEC Ada, the value of *small* in a fixed point representation clause can be any value in the range $2.0^{-62} \ .. \ 2.0^{31}$.

**Notes:**

13    A size specification is allowed for an access, task, or fixed point type, whether or not another form of length clause is also given for the type.

14    What is considered to be part of the storage reserved for a collection or for an activation of a task is implementation-dependent. The control afforded by length clauses is therefore relative to the implementation conventions. For example, the language does not define whether the storage reserved for an activation of a task includes any storage needed for the collection associated with an access type declared within the task body. Neither does it define the method of allocation for objects denoted by values of an access type. For example, the space allocated could be on a stack; alternatively, a general dynamic allocation scheme or fixed storage could be used.

The DEC Ada run-time reference manuals discuss task and access type storage and storage allocation in more detail.

15    The objects allocated in a collection need not have the same size if the designated type is an unconstrained array type or an unconstrained type with discriminants. Note also that the allocator itself may require some space for internal tables and links. Hence a length clause for the collection of an access type does not always give precise control over the maximum number of allocated objects.

16    **Examples:**

```
--  assumed declarations:

type MEDIUM is range 0 .. 65000;
type SHORT  is delta 0.01 range -100.0 .. 100.0;
type DEGREE is delta 0.1  range -360.0 .. 360.0;

BYTE : constant := 8;
PAGE : constant := 2000;

--  length clauses:

for COLOR'SIZE  use 1*BYTE;  --  see 3.5.1
for MEDIUM'SIZE use 2*BYTE;
for SHORT'SIZE  use 15;

for CAR_NAME'STORAGE_SIZE use  --  approximately 2000 cars
          2000*((CAR'SIZE/SYSTEM.STORAGE_UNIT) + 1);
```

---

[5] See also Appendix G, AI-00099.

```
for KEYBOARD_DRIVER'STORAGE_SIZE use 1*PAGE;

for DEGREE'SMALL use 360.0/2**(SYSTEM.STORAGE_UNIT - 1);
```

17 **Notes on the examples:**

In the length clause for SHORT, fifteen bits is the minimum necessary, since the type definition requires SHORT'SMALL = 2.0^{-7} and SHORT'MANTISSA = 14. The length clause for DEGREE forces the model numbers to exactly span the range of the type.

18 **References:** access type 3.8, allocator 4.8, allow 1.6, array type 3.6, attribute 4.1.4, collection 3.8, composite type 3.3, constraint 3.3, delta of a fixed point type 3.5.9, derived type 3.4, designate 3.8, elaboration 3.9, entity 3.1, evaluation 4.5, expression 4.4, first named subtype 13.1, fixed point type 3.5.9, index subtype 3.6, integer type 3.5.4, must 1.6, numeric type 3.5, object 3.2, real type 3.5.6, record type 3.7, small of a fixed point type 3.5.10, static constraint 4.9, static expression 4.9, static subtype 4.9, storage unit 13.7, subcomponent 3.3, system package 13.7, task 9, task activation 9.3, task specification 9.1, task type 9.2, type 3.3, unconstrained array type 3.6

component 3.6 3.7, constraint_error exception 11.1, discrete type 3.5, enumeration type 3.5.1, fixed point type declaration 3.5.9, length clause 13.2, object 3.2, pragma task_storage 13.2a, range of a fixed point type 3.5.9, representation clause 13.1

13.2a The Pragma Task_Storage

DEC Ada provides the pragma TASK_STORAGE to allow the specification of additional storage, called the *guard area*, for each task activation.

See the DEC Ada run-time reference manuals for more information about the guard area.

The form of this pragma is as follows:

```
pragma TASK_STORAGE([TASK_TYPE =>] simple_name,
    [TOP_GUARD =>] static_simple_expression);
```

The simple expression must be a static expression of some integer type. Its value specifies the additional number of storage units to be allocated as the guard area. The value is rounded up to an appropriate boundary. If the value is zero, then it has the following effect:

- On OpenVMS VAX systems, no guard area is created.

- On Alpha systems, a minimal guard area is created.

If the value is less than zero, the pragma is ignored and a default guard area is created.

A pragma TASK_STORAGE is allowed anywhere that a task storage size specification is allowed for the named task type: the pragma and the declaration of the task type to which the pragma applies must *both* occur immediately within the same declarative part or package specification. The type declaration *must* occur before the pragma. However, a pragma TASK_STORAGE can precede or follow any existing task storage size specification.

Example:

```
-- guard area will be 3 pages; activation size
-- (storage size) will be the default
task type EVEN_GUARD is
   . . .
end EVEN_GUARD;
pragma TASK_STORAGE (TASK_TYPE => EVEN_GUARD,
                     TOP_GUARD => 3*PAGE_SIZE);

-- guard area will be rounded up;
-- activation size (storage size) will be 10 pages
task type ROUND_IT is
   . . .
end ROUND_IT;
pragma TASK_STORAGE (TASK_TYPE => ROUND_IT,
                     TOP_GUARD => 1000);
for ROUND_IT'STORAGE_SIZE use 10*PAGE_SIZE;
```

References: allow 1.6, declarative part 3.9, integer type 3.5.4, package specification 7.1, pragma 2.8, simple expression 4.9, simple name 4.1, static expression 4.9, storage unit 13.7, task activation 9.3, task storage size specification 13.2, task type 9.2, task type declaration 9.1

13.2b The Pragma Main_Storage (OpenVMS VAX Systems Only)

DEC Ada provides the pragma MAIN_STORAGE to allow a fixed-size stack and stack storage areas to be specified for the task associated with a main program. Working storage can also be specified for the task activation, and additional storage, or guard pages, can be specified to provide protection against storage overflow during task execution of non-Ada code. The form of this pragma is as follows:

```
pragma MAIN_STORAGE(
    main_storage_option [, main_storage_option]);

main_storage_option :=
    [WORKING_STORAGE => ] static_simple_expression
  | [TOP_GUARD => ] static_simple_expression
```

The simple expression given for a main storage option must be a nonnegative static expression of some integer type. Its value specifies either the number of storage units (bytes) to be allocated for the main task stack working storage area or the number of storage units to be allocated as guard pages. If positional association is used, the first or only option is the working storage option. The second option is the top guard option.

For both the WORKING_STORAGE and TOP_GUARD options, the value specified is rounded up to an integral number of pages (where one page is 512 bytes). If the value is zero for WORKING_STORAGE, a default size is assumed. If the value is zero for TOP_GUARD, no guard pages are provided. If the value is less than zero for either WORKING_STORAGE or TOP_GUARD, the pragma is ignored.

A pragma MAIN_STORAGE is only allowed in the outermost declarative part of a library subprogram. At most one such pragma is allowed for a subprogram. This pragma has an effect only when the subprogram to which it applies is used as a main program.

Note:

When a program that specifies a pragma MAIN_STORAGE is run on an OpenVMS VAX system, the main stack is allocated in P0 space (rather than in the default P1 space). See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

Example:

```
procedure MAIN_PROGRAM is
pragma MAIN_STORAGE (WORKING_STORAGE => 10*512,
                     TOP_GUARD       => 1000);
begin
    . . .
end MAIN_PROGRAM;
```

In this example, the working storage for procedure MAIN_PROGRAM will be 10 pages. The guard storage will be 2 pages (1000 bytes rounds up to 2*512 bytes). The stack for the environment task associated with procedure MAIN_PROGRAM will be fixed. If the procedure is run on an OpenVMS VAX system, the stack for the environment task will be allocated in P0 space.

References: allow 1.6, declarative part 3.9, environment task 10.1, integer type 3.5.4, library unit 10.1, main program 10.1, positional parameter association 6.4, pragma 2.8, simple expression 4.4, static expression 4.9, storage unit 13.7, subprogram 6

13.3 Enumeration Representation Clauses

1 An enumeration representation clause specifies the internal codes for the literals of the enumeration type that is named in the clause.

2 ```
enumeration_representation_clause ::=
    for type_simple_name use aggregate;
```
[6]

3  The aggregate used to specify this mapping is written as a one-dimensional aggregate, for which the index subtype is the enumeration type and the component type is *universal_integer.*

4  All literals of the enumeration type must be provided with distinct integer codes, and all choices and component values given in the aggregate must be static. The integer codes specified for the enumeration type must satisfy the predefined ordering relation of the type.

In DEC Ada, each component value for an integer code must have a value in the range MIN_INT .. MAX_INT. Signed representation is used if any value given in an enumeration representation clause is negative.

5  **Example:**
```
type MIX_CODE is (ADD, SUB, MUL, LDA, STA, STZ);

for MIX_CODE use
   (ADD => 1, SUB => 2, MUL => 3,
    LDA => 8, STA => 24, STZ => 33);
```

Notes:

6 The attributes SUCC, PRED, and POS are defined even for enumeration types with a noncontiguous representation; their definition corresponds to the (logical) type declaration and is not affected by the enumeration representation clause. In the example, because of the need to avoid the omitted values, these functions are likely to be less efficiently implemented than they could be in the absence of a representation clause. Similar considerations apply when such types are used for indexing.

7 **References:** aggregate 4.3, array aggregate 4.3.2, array type 3.6, attribute of an enumeration type 3.5.5, choice 3.7.3, component 3.3, enumeration literal 3.5.1, enumeration type 3.5.1, function 6.5, index 3.6, index subtype 3.6, literal 4.2, ordering relation of an enumeration type 3.5.1, representation clause 13.1, simple name 4.1, static expression 4.9, type 3.3, type declaration 3.3.1, universal_integer type 3.5.4

system.max_int 13.7, system.min_int 13.7

---

[6] See also Appendix G, AI-00422.

13.4 Record Representation Clauses

1 A record representation clause specifies the storage representation of records, that is, the order, position, and size of record components (including discriminants, if any).

2
```
record_representation_clause ::=
    for type_simple_name use
        record [alignment_clause]
            {component_clause}
        end record;

alignment_clause ::= at mod static_simple_expression;

component_clause ::=
    component_name at static_simple_expression
        range static_range;
```
[7]

3 The simple expression given after the reserved words **at mod** in an alignment clause, or after the reserved word **at** in a component clause, must be a static expression of some integer type. If the bounds of the range of a component clause are defined by simple expressions, then each bound of the range must be defined by a static expression of some integer type, but the two bounds need not have the same integer type.

4 An alignment clause forces each record of the given type to be allocated at a starting address that is a multiple of the value of the given expression (that is, the address modulo the expression must be zero). An implementation may place restrictions on the allowable alignments.

DEC Ada allows the simple expression in an alignment clause to have a value of 2^n, where

   0 <= n <= 9 on OpenVMS VAX systems
   0 <= n <= 16 on OpenVMS Alpha systems
   0 <= n <= 3 on Digital UNIX systems

In other words, the simple expression must be an integer in the range (1 .. 512, 1 .. 65536, or 1 .. 8) that is also a power of 2. The allocations then occur at addresses that are a multiple of the simple expression (a value of 2 aligns the data on a 2-byte boundary, a value of 4 aligns the data on a 4-byte boundary, and so on).

There are no other restrictions on record type alignments. See the DEC Ada run-time reference manuals for more information on how record objects are allocated.

---

[7] See also Appendix G, AI-00422.

5   A component clause specifies the *storage place* of a component, relative to the start of the record. The integer defined by the static expression of a component clause is a relative address expressed in storage units. The range defines the bit positions of the storage place, relative to the storage unit. The first storage unit of a record is numbered zero. The first bit of a storage unit is numbered zero. The ordering of bits in a storage unit is machine-dependent and may extend to adjacent storage units. (For a specific machine, the size in bits of a storage unit is given by the configuration-dependent named number SYSTEM.STORAGE_UNIT.) Whether a component is allowed to overlap a storage boundary, and if so, how, is implementation-defined.

In DEC Ada, the size of a storage unit (SYSTEM.STORAGE_UNIT) is eight bits (one byte). If the number of bits specified by the range is sufficient for the component subtype, the requested size and placement of the field is observed (and overlaps storage boundaries if necessary). Otherwise, the specification is illegal. For a component of a discrete or fixed point type, the number of bits must be as follows:

- On OpenVMS VAX systems, the number of bits must be between 1 and 32 for all types.

- On Alpha systems, the number of bits must be between 1 and 64 for a discrete type; between 1 and 32 for a fixed point type.

For a component of any other type, the size must not exceed the actual size of the component. See the DEC Ada run-time reference manuals for information about determining the number of bits that are sufficient for any given subtype.

In DEC Ada, component values of integer types may be biased when a component clause requires a very small component storage space. Each value stored is the unsigned quantity formed by subtracting the value COMPONENT_SUBTYPE'FIRST from the original value. See the DEC Ada run-time reference manuals for more information.

Component clauses in DEC Ada are restricted as follows:

- Any component that is not packable must be allocated on a byte boundary. Components that are packable can be allocated without restriction. See the DEC Ada run-time reference manuals for a definition and description of packable components.

- Any representation clause or pragma that applies to the component type still applies (affects the alignment and size of the component).

6   At most one component clause is allowed for each component of the record type,
    including for each discriminant (component clauses may be given for some, all,
    or none of the components). If no component clause is given for a component,
    then the choice of the storage place for the component is left to the compiler.
    If component clauses are given for all components, the record representation
    clause completely specifies the representation of the record type and must be
    obeyed exactly by the compiler.

    In DEC Ada, components named in a component clause are allocated first.
    Unnamed components are allocated in the order in which they are written in
    the record type declaration. Variants can be overlapped. If the pragma PACK
    is specified, packed allocation rules are used. Otherwise, unpacked allocation
    is used. If the pragma COMPONENT_ALIGNMENT is specified, its alignment
    rules (see 13.1a) are used. Otherwise, default alignments are used.

7   Storage places within a record variant must not overlap, but overlap of the
    storage for distinct variants is allowed. Each component clause must allow for
    enough storage space to accommodate every allowable value of the component.
    A component clause is only allowed for a component if any constraint on this
    component or on any of its subcomponents is static.[8]

8   An implementation may generate names that denote implementation-
    dependent components (for example, one containing the offset of another
    component). Such implementation-dependent names can be used in record
    representation clauses (these names need not be simple names; for example,
    they could be implementation-dependent attributes).

    DEC Ada generates no implementation-dependent components or names.

9   **Example:**

```
WORD : constant := 4;  --  storage unit is byte,
                       --   4 bytes per word

type STATE       is (A, M, W, P);
type MODE        is (FIX, DEC, EXP, SIGNIF);

type BYTE_MASK  is array (0 .. 7) of BOOLEAN;
type STATE_MASK is array (STATE)  of BOOLEAN;
type MODE_MASK  is array (MODE)   of BOOLEAN;

pragma PACK (BYTE_MASK);  -- in DEC Ada these must be packed
pragma PACK (STATE_MASK); -- for the alignment to work
pragma PACK (MODE_MASK);
```

---

[8] See also Appendix G, AI-00132.

```
type PROGRAM_STATUS_WORD is
   record
      SYSTEM_MASK      : BYTE_MASK;
      PROTECTION_KEY   : INTEGER range 0 .. 3;
      MACHINE_STATE    : STATE_MASK;
      INTERRUPT_CAUSE  : INTERRUPTION_CODE;
      ILC              : INTEGER range 0 .. 3;
      CC               : INTEGER range 0 .. 3;
      PROGRAM_MASK     : MODE_MASK;
      INST_ADDRESS     : ADDRESS;
   end record;

for PROGRAM_STATUS_WORD use
   record at mod 8;
      SYSTEM_MASK     at 0*WORD range 0  .. 7;   -- bits 8,9
      PROTECTION_KEY  at 0*WORD range 10 .. 11;  -- unused
      MACHINE_STATE   at 0*WORD range 12 .. 15;
      INTERRUPT_CAUSE at 0*WORD range 16 .. 31;
      ILC             at 1*WORD range 0  .. 1;   -- second word
      CC              at 1*WORD range 2  .. 3;
      PROGRAM_MASK    at 1*WORD range 4  .. 7;
      INST_ADDRESS    at 1*WORD range 8  .. 31;
   end record;

for PROGRAM_STATUS_WORD'SIZE use 8*SYSTEM.STORAGE_UNIT;
```

Note on the example:

10   The record representation clause defines the record layout. The length clause
      guarantees that exactly eight storage units are used.

The example assumes that the type ADDRESS is represented in 24 bits.

On OpenVMS systems, DEC Ada represents the type ADDRESS in 32 bits.

On Digital UNIX systems, DEC Ada represents the type ADDRESS in 64 bits.

Component Specification Example:
```
subtype S is INTEGER range 10 .. 13;

type REC is
    record
       X : S;
       Y : S;
       Z : S;
    end record;

for REC use
    record
       X at 0 range 0 .. 4;  -- legal: 4 bits are sufficient for
                             -- an unsigned representation

       Y at 0 range 5 .. 6;  -- legal: 2 bits are sufficient for
                             -- a biased representation
```

```
            Z at 0 range 7 .. 7;  -- illegal: 1 bit is not enough to
                                  -- represent an integer of subtype S
      end record;
```

Notes on the example:

The subtype declaration in this example implies an integer with a minimum
size of 4 bits. The component clause for X is legal because it requires at least
the minimum number of bits required for the integer subtype. The component
clause for Y is legal because it requires at least the minimum number of bits
required for a biased representation of the subtype. The component clause
for Z is illegal because it does not allow enough bits to represent the integer
subtype.

11 **References:** allow 1.6, attribute 4.1.4, constant 3.2.1, constraint 3.3, discriminant
3.7.1, integer type 3.5.4, must 1.6, named number 3.2, range 3.5, record component 3.7,
record type 3.7, simple expression 4.4, simple name 4.1, static constraint 4.9, static
expression 4.9, storage unit 13.7, subcomponent 3.3, system package 13.7, variant 3.7.3

component clause 13.4, component subtype 3.7, object 3.2, packable 13.1, pragma pack
13.1

13.5 Address Clauses

1 An address clause specifies a required address in storage for an entity.

2   address_clause ::= **for** simple_name **use at** simple_expression;

3 The expression given after the reserved word **at** must be of the type ADDRESS
defined in the package SYSTEM (see 13.7); this package must be named by a
with clause that applies to the compilation unit in which the address clause
occurs. The conventions that define the interpretation of a value of the type
ADDRESS as an address, as an interrupt level, or whatever it may be, are
implementation-dependent. The allowed nature of the simple name and the
meaning of the corresponding address are as follows:

4   (a) Name of an object: the address is that required for the object (variable
           or constant). [9]

5   (b) Name of a subprogram, package, or task unit: the address is that
           required for the machine code associated with the body of the program
           unit. [10]

6   (c) Name of a single entry: the address specifies a hardware interrupt to
           which the single entry is to be linked.

---

[9] See also Appendix G, AI-00263.
[10] See also Appendix G, AI-00336.

7   If the simple name is that of a single task, the address clause is understood to refer to the task unit and not to the task object. In all cases, the address clause is only legal if exactly one declaration with this identifier occurs earlier, immediately within the same declarative part, package specification, or task specification. A name declared by a renaming declaration is not allowed as the simple name.

In DEC Ada, the simple name must be the name of an object or imported subprogram. On Digital UNIX systems, the simple name can also be the name of a single entry.

Address clauses are not allowed in combination with any of the DEC Ada pragmas for importing or exporting objects. If used in such cases, the pragma involved is ignored.

The rules for specifying an address clause for an imported subprogram are as follows:

- The subprogram must be specified with a pragma INTERFACE. It may also be specified with an import pragma:

  IMPORT_FUNCTION
  IMPORT_PROCEDURE
  IMPORT_VALUED_PROCEDURE
  INTERFACE_NAME

- If the subprogram is specified in an import pragma, a null string ("") must be specified for the external designator (see 13.9a).

- The address clause must be specified after the pragma INTERFACE. If a DEC Ada import pragma is also specified, the order of the address clause and the import pragma is not significant.

8   Address clauses should not be used to achieve overlays of objects or overlays of program units. Nor should a given interrupt be linked to more than one entry. Any program using address clauses to achieve such effects is erroneous. [11]

9   **Example:**
```
for CONTROL use at 16#0020#;  --  assuming that SYSTEM.ADDRESS
                              --  is an integer type
```

---

[11] See also Appendix G, AI-00292 and AI-00379.

Notes:

10   The above rules imply that if two subprograms overload each other and are visible at a given point, an address clause for any of them is not legal at this point. Similarly if a task specification declares entries that overload each other, they cannot be interrupt entries. The syntax does not allow an address clause for a library unit. An implementation may provide pragmas for the specification of program overlays.

In DEC Ada, the alignment required for the address is checked against the alignment given for the type in the following cases:

- If an address clause is specified for a record object whose type has been declared with an alignment clause.

- If an address clause is specified for a record or array object whose type has been specified in a pragma COMPONENT_ALIGNMENT, and the alignment choice is either COMPONENT_SIZE or COMPONENT_SIZE_4.

If the two alignments are incompatible, the exception PROGRAM_ERROR is raised.

The same check applies to a type that contains a component whose type has been declared with an alignment clause or a pragma COMPONENT_ ALIGNMENT (the alignment of the component forces the alignment of the containing type).

In DEC Ada, the usual implicit initialization associated with a type is performed, even when an object of the type is declared with an address clause. For example, access values are initialized to null, and record components can also receive initial values. See the DEC Ada run-time reference manuals for more information.

11   **References:** address predefined type 13.7, apply 10.1.1, compilation unit 10.1, constant 3.2.1, entity 3.1, entry 9.5, erroneous 1.6, expression 4.4, library unit 10.1, name 4.1, object 3.2, package 7, pragma 2.8, program unit 6, reserved word 2.9, simple expression 4.4, simple name 4.1, subprogram 6, subprogram body 6.3, system package 13.7, task body 9.1, task object 9.2, task unit 9, type 3.3, variable 3.2.1, with clause 10.1.1

declarative part 3.9, forcing occurrence 13.1, imported subprogram 13.9a 13.9a.1.1 package specification 7.1

13.5.1 Interrupts

On OpenVMS systems, DEC Ada does not support interrupts as defined in this section. Instead, DEC Ada provides the pragma AST_ENTRY and the AST_ENTRY attribute as alternative mechanisms for handling asynchronous interrupts from the OpenVMS operating system. See section 9.12a for more information on this pragma and attribute.

On Digital UNIX systems, DEC Ada supports interrupts as defined in this section to allow some Digital UNIX signals to be associated with task entry calls.

1 An address clause given for an entry associates the entry with some device that may cause an interrupt; such an entry is referred to in this section as an *interrupt entry.* If control information is supplied upon an interrupt, it is passed to an associated interrupt entry as one or more parameters of mode **in**; only parameters of this mode are allowed.

2 An interrupt acts as an entry call issued by a hardware task whose priority is higher than the priority of the main program, and also higher than the priority of any user-defined task (that is, any task whose type is declared by a task unit in the program). The entry call may be an ordinary entry call, a timed entry call, or a conditional entry call, depending on the kind of interrupt and on the implementation.

3 If a select statement contains both a terminate alternative and an accept alternative for an interrupt entry, then an implementation may impose further requirements for the selection of the terminate alternative in addition to those given in section 9.4.

4 **Example:**
```
task INTERRUPT_HANDLER is
   entry DONE;
   for DONE use at 16#40#;   -- assuming that SYSTEM.ADDRESS
                             -- is an integer type
end INTERRUPT_HANDLER;
```

Notes:

5 Interrupt entry calls need only have the semantics described above; they may be implemented by having the hardware directly execute the appropriate accept statements.

6 Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an accept statement executed in response to an interrupt takes precedence over ordinary, user-defined tasks, and can be executed without first invoking a scheduling action.

7   One of the possible effects of an address clause for an interrupt entry is to specify the priority of the interrupt (directly or indirectly). Direct calls to an interrupt entry are allowed.

8   **References:** accept alternative 9.7.1, accept statement 9.5, address predefined type 13.7, allow 1.6, conditional entry call 9.7.2, entry 9.5, entry call 9.5, mode 6.1, parameter of a subprogram 6.2, priority of a task 9.8, select alternative 9.7.1, select statement 9.7, system package 13.7, task 9, terminate alternative 9.7.1, timed entry call 9.7.3

13.6 Change of Representation

1   At most one representation clause is allowed for a given type and a given aspect of its representation. Hence, if an alternative representation is needed, it is necessary to declare a second type, derived from the first, and to specify a different representation for the second type. [12]

2   **Example:**
```
--  PACKED_DESCRIPTOR and DESCRIPTOR are two different
--  types with identical characteristics, apart from
--  their representation

type DESCRIPTOR is
   record
      --  components of a descriptor
   end record;

type PACKED_DESCRIPTOR is new DESCRIPTOR;

for PACKED_DESCRIPTOR use
   record
      --  component clauses for some or for all components
   end record;
```

3   Change of representation can now be accomplished by assignment with explicit type conversions:
```
D : DESCRIPTOR;
P : PACKED_DESCRIPTOR;

P := PACKED_DESCRIPTOR(D);  --  pack D
D := DESCRIPTOR(P);         --  unpack P
```

4   **References:** assignment 5.2, derived type 3.4, type 3.3, type conversion 4.6, type declaration 3.1, representation clause 13.1

---

[12] See also Appendix G, AI-00040 and AI-00138.

13.7 The Package System

1   For each implementation there is a predefined library package called SYSTEM which includes the definitions of certain configuration-dependent characteristics. The specification of the package SYSTEM is implementation-dependent and must be given in Appendix F. The visible part of this package must contain at least the following declarations.

DEC Ada additions to the package SYSTEM are described in sections 13.7a and 13.7b.

A composite DEC Ada specification of the package SYSTEM is included in Appendix F.

2
```ada
package SYSTEM is
    type ADDRESS is implementation_defined;
    type NAME    is implementation_defined_enumeration_type;

    SYSTEM_NAME  : constant NAME := implementation_defined;

    STORAGE_UNIT : constant := implementation_defined;
    MEMORY_SIZE  : constant := implementation_defined;

    --  System-Dependent Named Numbers:

    MIN_INT      : constant := implementation_defined;
    MAX_INT      : constant := implementation_defined;
    MAX_DIGITS   : constant := implementation_defined;
    MAX_MANTISSA : constant := implementation_defined;
    FINE_DELTA   : constant := implementation_defined;
    TICK         : constant := implementation_defined;
```

```
                -- Other System-Dependent Declarations

        subtype PRIORITY is INTEGER range implementation_defined;
        ...
end SYSTEM;13
```

³ The type ADDRESS is the type of the addresses provided in address clauses; it
is also the type of the result delivered by the attribute ADDRESS. Values of the
enumeration type NAME are the names of alternative machine configurations
handled by the implementation; one of these is the constant SYSTEM_NAME.
The named number STORAGE_UNIT is the number of bits per storage unit;
the named number MEMORY_SIZE is the number of available storage units in
the configuration; these named numbers are of the type *universal_integer.*

In DEC Ada, values of the type ADDRESS refer to addresses in virtual address
space.

⁴ An alternative form of the package SYSTEM, with given values for any of
SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE, can be obtained
by means of the corresponding pragmas. These pragmas are only allowed
at the start of a compilation, before the first compilation unit (if any) of the
compilation.

⁵   **pragma** SYSTEM_NAME(enumeration_literal);

⁶ The effect of the above pragma is to use the enumeration literal with the
specified identifier for the definition of the constant SYSTEM_NAME. This
pragma is only allowed if the specified identifier corresponds to one of the
literals of the type NAME.

In DEC Ada, the enumeration literals for the type NAME are system-specific,
as follows:

System	Enumeration literal
OpenVMS VAX	VAX_VMS
OpenVMS Alpha	OpenVMS_AXP
Digital UNIX	DEC_OSF1_AXP

On OpenVMS and Digital UNIX systems, all literals are defined in the type
SYSTEM.NAME.

⁷   **pragma** STORAGE_UNIT(numeric_literal);

---

¹³ See also Appendix G, AI-00045 and
AI-00355.

8   The effect of the above pragma is to use the value of the specified numeric literal for the definition of the named number STORAGE_UNIT.

In DEC Ada, the value given for STORAGE_UNIT must be 8 (bits).

9   **pragma** MEMORY_SIZE(numeric_literal);

10   The effect of the above pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE.

In DEC Ada, the number given for MEMORY_SIZE must be in the range 0 .. MAX_INT. It replaces the default value (MAX_INT). DEC Ada does not provide support for checking or ensuring that the given size is not exceeded.

11   The compilation of any of these pragmas causes an implicit recompilation of the package SYSTEM. Consequently any compilation unit that names SYSTEM in its context clause becomes obsolete after this implicit recompilation. An implementation may impose further limitations on the use of these pragmas. For example, an implementation may allow them only at the start of the first compilation, when creating a new program library.

DEC Ada imposes no further limitations on these pragmas. To reduce the amount of recompilation required, DEC Ada identifies those units that have a real dependence on the values affected by these pragmas. Only such units must be recompiled. In particular, predefined DEC Ada packages do not depend on the values affected by these pragmas, and none requires recompilation if these pragmas are used.

Note:

12   It is a consequence of the visibility rules that a declaration given in the package SYSTEM is not visible in a compilation unit unless this package is mentioned by a with clause that applies (directly or indirectly) to the compilation unit.

13   **References:** address clause 13.5, apply 10.1.1, attribute 4.1.4, compilation unit 10.1, declaration 3.1, enumeration literal 3.5.1, enumeration type 3.5.1, identifier 2.3, library unit 10.1, must 1.6, named number 3.2, number declaration 3.2.2, numeric literal 2.4, package 7, package specification 7.1, pragma 2.8, program library 10.1, type 3.3, visibility 8.3, visible part 7.2, with clause 10.1.1

dependence between compilation units 10.3, erroneous 1.6, exception 11, expression 4.4, integer type 3.5.4, range 3.5, system.max_int 13.7

13.7.1 System-Dependent Named Numbers

1 Within the package SYSTEM, the following named numbers are declared. The numbers FINE_DELTA and TICK are of the type *universal_real*; the others are of the type *universal_integer.*

2   MIN_INT   The smallest (most negative) value of all predefined integer types.

In DEC Ada on OpenVMS VAX systems, the value for MIN_INT is -2^{31}.

In DEC Ada on Alpha systems, the value for MIN_INT is -2^{63}.

3   MAX_INT   The largest (most positive) value of all predefined integer types.

In DEC Ada on OpenVMS VAX systems, the value for MAX_INT is $2^{31} - 1$.

In DEC Ada on Alpha systems, the value for MAX_INT is $2^{63} - 1$.

4   MAX_DIGITS   The largest value allowed for the number of significant decimal digits in a floating point constraint.

In DEC Ada on OpenVMS VAX systems, the value for MAX_DIGITS is 33.

In DEC Ada on Alpha systems, the value for MAX_DIGITS is 15.

5   MAX_MANTISSA   The largest possible number of binary digits in the mantissa of model numbers of a fixed point subtype.

In DEC Ada, the value for MAX_MANTISSA is 31.

6   FINE_DELTA   The smallest delta allowed in a fixed point constraint that has the range constraint $-1.0\ ..\ 1.0$.

In DEC Ada, the value for FINE_DELTA is 2.0^{-31}.

7   TICK   The basic clock period, in seconds. [14]

In DEC Ada on OpenVMS VAX systems, the value for TICK is 10.0^{-2} (or 10 milliseconds). This value corresponds to the clock interval provided by the time-related OpenVMS system services.

---

[14] See also Appendix G, AI-00201 and AI-00366.

In DEC Ada on Alpha systems, the value for TICK is 10.0^{-3} (or 1 millisecond). On OpenVMS Alpha systems, this value corresponds to the clock interval provided by the time-related OpenVMS system services. On Digital UNIX systems, this value corresponds to the resolution of the Digital UNIX system clock.

8 **References:** allow 1.6, delta of a fixed point constraint 3.5.9, fixed point constraint 3.5.9, floating point constraint 3.5.7, integer type 3.5.4, model number 3.5.6, named number 3.2, package 7, range constraint 3.5, system package 13.7, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6

13.7.2 Representation Attributes

1 The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate *representation attributes.* These attributes are described below.

2 For any object, program unit, label, or entry X:

3   X′ADDRESS   Yields the address of the first of the storage units allocated to X. For a subprogram, package, task unit or label, this value refers to the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, the value refers to the corresponding hardware interrupt. The value of this attribute is of the type ADDRESS defined in the package SYSTEM. [15]

For an object that is a variable in DEC Ada, the value is the actual address of the variable (which may be statically or dynamically allocated). This attribute forces a variable to be allocated in memory. It causes the variable to be marked as volatile for the duration of the block statement or body containing the use of the attribute. If the location of the variable is not byte-aligned, the value is the address of the lowest byte that holds part or all of the variable. For an object that is a constant, the value is the address of the constant value in memory. However, two occurrences of C′ADDRESS, where C denotes a constant, may or may not yield the same address value. For an object

---

[15] See also Appendix G, AI-00305.

that is a named number, the value is zero (ADDRESS_ ZERO).

For an access object, X.**all** ' ADDRESS is the address of the designated object. X.**all** ' ADDRESS is subject to an ACCESS_CHECK for the designated object. For a record component, X.C ' ADDRESS is subject to a DISCRIMINANT_CHECK for an object in a variant part. For an array component or slice, X(I) ' ADDRESS or X(I1 .. I2) ' ADDRESS is subject to an INDEX_ CHECK for the denoted component or slice.

For program units that are generic units, task units, or package units, the value is zero (ADDRESS_ ZERO). For program units that are imported or exported subprograms, the value is the same as the address that is imported or exported. (See section 13.9a.1.1 for information on the pragmas IMPORT_ FUNCTION, IMPORT_PROCEDURE, and IMPORT_ VALUED_PROCEDURE. See section 13.9a.1.3 for information on the pragmas EXPORT_FUNCTION, EXPORT_PROCEDURE, and EXPORT_VALUED_ PROCEDURE. See section 13.9b for information on the pragma INTERFACE_NAME.) For subprograms that are not imported or exported, the value is zero (ADDRESS_ZERO).

For labels or entries, the value is zero (ADDRESS_ ZERO).

4   For any type or subtype X, or for any object X:

5   X ' SIZE   Applied to an object, yields the number of bits
                                    allocated to hold the object. Applied to a type or
                                    subtype, yields the minimum number of bits that is
                                    needed by the implementation to hold any possible
                                    object of this type or subtype. The value of this
                                    attribute is of the type *universal_integer.*

                                    For a type or a subtype in DEC Ada, the value is
                                    limited to values in the range $0 .. 2^{31}-1$. The exception
                                    CONSTRAINT_ERROR is raised for values outside
                                    this range.

                                    For an object that is a variable or a constant in DEC
                                    Ada, the value is its size in bits.

For an object that is a named number, the value is zero.

For a record component, X.C'SIZE is subject to a DISCRIMINANT_CHECK for an object in a variant part.

For an array component or slice, X(I)'SIZE or X(I1 .. I2)'SIZE is subject to an INDEX_CHECK for the denoted component or slice.

6 For the above two representation attributes, if the prefix is the name of a function, the attribute is understood to be an attribute of the function (not of the result of calling the function). Similarly, if the type of the prefix is an access type, the attribute is understood to be an attribute of the prefix (not of the designated object: attributes of the latter can be written with a prefix ending with the reserved word **all**). [16]

For any type or subtype X:

X'MACHINE_SIZE Yields the total number of machine bits to be allocated for objects of the type or subtype. This value takes into account any padding bits used by DEC Ada when allocating storage for an object. The value of this attribute is of the type *universal_integer*.

The value is always a multiple of the size of a storage unit (in DEC Ada, this size is eight (bits)). The value is limited to the range $0 .. 2^{31}-1$. The exception CONSTRAINT_ERROR is raised for values outside this range.

See the DEC Ada run-time reference manuals for more discussion of this attribute and for more information on DEC Ada storage allocation.

For any object X:

X'BIT   Yields the bit offset within the storage unit (byte) that contains the first bit of the storage allocated for the object. The value of this attribute is of the type *universal_integer*, and is always in the range 0 .. 7.

---

[16] See also Appendix G, AI-00012 and AI-00015.

For an object that is a variable or a constant allocated in a register, the value is zero. (The use of this attribute does not force the allocation of a variable to memory.)

For an object that is a formal parameter, this attribute applies to either the matching actual parameter or to a copy of the matching actual parameter.

For an access object, the value is zero (in the absence of CONSTRAINT_ERROR). X.**all**'BIT is subject to an ACCESS_CHECK for the designated object.

For a record component, X.C'BIT is subject to a DISCRIMINANT_CHECK for a component in a variant part.

For an array component or slice, X(I)'BIT or X(I1 .. I2)'BIT is subject to an INDEX_CHECK for the denoted component or slice.

7   For any component C of a record object R: [17]

8   R.C'POSITION   Yields the offset, from the start of the first storage unit occupied by the record, of the first of the storage units occupied by C. This offset is measured in storage units. The value of this attribute is of the type *universal_integer.* [18]

9   R.C'FIRST_BIT   Yields the offset, from the start of the first of the storage units occupied by C, of the first bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer.*

10   R.C'LAST_BIT   Yields the offset, from the start of the first of the storage units occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer.*

11 For any access type or subtype T:

12   T'STORAGE_SIZE Yields the total number of storage units reserved for the collection associated with the base type of T. The value of this attribute is of the type *universal_integer.*

13 For any task type or task object T:

---

[17] See also Appendix G, AI-00258.
[18] See also Appendix G, AI-00362 and AI-00503.

14   T'STORAGE_SIZE   Yields the number of storage units reserved for each activation of a task of the type T or for the activation of the task object T. The value of this attribute is of the type *universal_integer.*

Notes:

15   For a task object X, the attribute X'SIZE gives the number of bits used to hold the object X, whereas X'STORAGE_SIZE gives the number of storage units allocated for the activation of the task designated by X. For a formal parameter X, if parameter passing is achieved by copy, then the attribute X'ADDRESS yields the address of the local copy; if parameter passing is by reference, then the address is that of the actual parameter.

The attribute X'MACHINE_SIZE gives the size that is used for a variable of the type or subtype. It does not give the size that can be used for a component of that type or subtype.

The machine size of a type or subtype can be influenced by representation clauses or pragmas, unlike the size of a type or subtype, which is independent of representation clauses or pragmas. The machine size of a base type can be less than, equal to, or greater than the size of that same base type. See the DEC Ada run-time reference manuals for examples and additional discussion.

The machine size of a type or subtype determines the number of bits that are read or written to external files in input-output operations for elements of that type or subtype (see chapter 14 for information on input-output operations).

16   **References:** access subtype 3.8, access type 3.8, activation 9.3, actual parameter 6.2, address clause 13.5, address predefined type 13.7, attribute 4.1.4, base type 3.3, collection 3.8, component 3.3, entry 9.5, formal parameter 6.1 6.2, label 5.1, object 3.2, package 7, package body 7.1, parameter passing 6.2, program unit 6, record object 3.7, statement 5, storage unit 13.7, subprogram 6, subprogram body 6.3, subtype 3.3, system predefined package 13.7, task 9, task body 9.1, task object 9.2, task type 9.2, task unit 9, type 3.3, universal_integer type 3.5.4

access object 3.8, array component 3.6, block statement 5.6, body 3.9, constant 3.2.1, constraint_error exception 11.1, designated object 3.8, exported subprogram 13.9a.1.3, generic unit 12 12.1, integer type 3.5.4, named number 3.2, package 7, range 3.5, record component 3.7, slice 4.1.2, subtype 3.3, system.address_zero 13.7a.1, system.max_int 13.7, type 3.3, variable 3.2.1, variant part 3.7.3

13.7.3 Representation Attributes of Real Types

1   For every real type or subtype T, the following machine-dependent attributes are defined, which are not related to the model numbers. Programs using these attributes may thereby exploit properties that go beyond the minimal properties associated with the numeric type (see section 4.5.7 for the rules defining the accuracy of operations with real operands). Precautions must therefore be taken when using these machine-dependent attributes if portability is to be ensured.

2   For both floating point and fixed point types:

3   T'MACHINE_ROUNDS   Yields the value TRUE if every predefined arithmetic operation on values of the base type of T either returns an exact result or performs rounding; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. [19]

In DEC Ada on OpenVMS VAX systems, this value is TRUE for floating point types. It is FALSE for any fixed point type where the value of *small* is a power of two. It is TRUE for all other fixed point types.

In DEC Ada on all other systems, this value is TRUE.

(In DEC Ada, the value of *small* is a power of two by default. A length clause can be used to achieve other values. See 13.2.)

4   T'MACHINE_OVERFLOWS   Yields the value TRUE if every predefined operation on values of the base type of T either provides a correct result, or raises the exception NUMERIC_ERROR in overflow situations (see 4.5.7); yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN.

In DEC Ada, this value is TRUE.

---
[19] See also Appendix G, AI-00263.

5   For floating point types, the following attributes provide characteristics of the underlying machine representation, in terms of the canonical form defined in section 3.5.7: [20]

6   T'MACHINE_RADIX   Yields the value of the *radix* used by the machine representation of the base type of T. The value of this attribute is of the type *universal_integer*.

In DEC Ada, this value is 2.

7   T'MACHINE_MANTISSA   Yields the number of digits in the *mantissa* for the machine representation of the base type of T (the digits are extended digits in the range 0 to T'MACHINE_RADIX − 1). The value of this attribute is of the type *universal_integer*.

For types based on the VAX hardware types, this value can be 24 (F_floating), 53 (G_floating), 56 (D_floating), or 113 (H_floating).

For types based on the IEEE hardware types, this value can be 24 (IEEE single float) or 53 (IEEE double float).

8   T'MACHINE_EMAX   Yields the largest value of *exponent* for the machine representation of the base type of T. The value of this attribute is of the type *universal_integer*.

For types based on the VAX hardware types, this value can be 127 (F_floating or D_floating), 1023 (G_floating), or 16,383 (H_floating).

For types based on the IEEE hardware types, this value can be 128 (IEEE single float) or 1024 (IEEE double float).

9   T'MACHINE_EMIN   Yields the smallest (most negative) value of *exponent* for the machine representation of the base type of T. The value of this attribute is of the type *universal_integer*.

---

[20] See also Appendix G, AI-00263.

> For types based on the VAX hardware types, this value can be –127 (F_floating or D_floating), –1023 (G_floating), or –16,383 (H_floating).
>
> For types based on the IEEE hardware types, this value can be –125 (IEEE single float) or –1021 (IEEE double float).

The DEC Ada values of the representation attributes for floating point numbers are listed in Appendix F. Definitions of the six floating point representations (F_floating, D_floating, G_floating, H_floating, IEEE single float, and IEEE double float) are given in section 3.5.7.

Note:

10 For many machines the largest machine representable number of type F is almost

```
(F'MACHINE_RADIX)**(F'MACHINE_EMAX),
```

11 and the smallest positive representable number is

```
F'MACHINE_RADIX ** (F'MACHINE_EMIN - 1)
```

12 **References:** arithmetic operator 4.5, attribute 4.1.4, base type 3.3, boolean predefined type 3.5.3, false boolean value 3.5.3, fixed point type 3.5.9, floating point type 3.5.7, model number 3.5.6, numeric type 3.5, numeric_error exception 11.1, predefined operation 3.3.3, radix 3.5.7, real type 3.5.6, subtype 3.3, true boolean value 3.5.3, type 3.3, universal_integer type 3.5.4

d_floating representation 3.5.7 3.5.7b, f_floating representation 3.5.7, g_floating representation 3.5.7 3.5.7b, h_floating representation 3.5.7, IEEE double float representation 3.5.7, IEEE single float representation 3.5.7

13.7a General DEC Ada Additions to the Package System

In addition to the language-required declarations in the package SYSTEM, DEC Ada declares the operations, constants, types, and subtypes described in the following sections. The information in these sections applies to all DEC Ada compilers.

13.7a.1 Address Type Declarations

In DEC Ada, ADDRESS is a private type for which the following operations are declared in the package SYSTEM:

```
type ADDRESS is private;

ADDRESS_ZERO : constant ADDRESS;
NO_ADDR      : constant ADDRESS;
NULL_ADDRESS : constant ADDRESS;

-- System-specific definition of ADDRESS_SIZE
--
---- On Digital UNIX systems:
ADDRESS_SIZE : constant := 64;
--
---- On OpenVMS systems:
ADDRESS_SIZE : constant := 32;

function "+" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
function "+" (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
function "-" (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
function "-" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;

-- Because ADDRESS is a private type
-- the functions "=" and "/=" are already available and
-- do not have to be explicitly defined
--
-- function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
   function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

generic
   type TARGET is private;
function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

generic
   type TARGET is private;
procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);
```

The addition, subtraction, and relational functions provide arithmetic and comparative operations for addresses. The generic subprograms FETCH_FROM_ADDRESS and ASSIGN_TO_ADDRESS provide operations for reading from or writing to a given address interpreted as having any desired type. The value of the deferred constant ADDRESS_ZERO corresponds to the first (machine) address.

In an instantiation of FETCH_FROM_ADDRESS or ASSIGN_TO_ADDRESS, the actual subtype corresponding to the formal type TARGET must be neither an unconstrained array type nor an unconstrained type with discriminants. If the actual subtype is a type with discriminants, the value fetched by a call of a function resulting from an instantiation of FETCH_FROM_ADDRESS is checked to ensure that the discriminants satisfy the constraints of the actual subtype. In any other case, no check is made.

Functions for converting values to and from the type ADDRESS are also provided, as follows:

```
function TO_ADDRESS  (X : INTEGER)             return ADDRESS;
function TO_ADDRESS  (X : UNSIGNED_LONGWORD)   return ADDRESS;
function TO_ADDRESS  (X : {universal_integer}) return ADDRESS;

function TO_INTEGER           (X : ADDRESS)    return INTEGER;
function TO_UNSIGNED_LONGWORD (X : ADDRESS)    return UNSIGNED_LONGWORD;
```

See section 13.7a.4 for information about the type SYSTEM.UNSIGNED_LONGWORD.

Note:

On OpenVMS and Digital UNIX systems, DEC Ada provides a predefined library package of integer-related address operations. This package is generic and can be instantiated for any integer type. It is provided to accommodate for differences among address representations across DEC Ada systems.

```
with SYSTEM;
with UNCHECKED_CONVERSION;
generic
   type INTEGER_TYPE is range <>;
package ADDRESS_OPERATIONS is

   function "+" (LEFT : SYSTEM.ADDRESS; RIGHT : INTEGER_TYPE)
      return SYSTEM.ADDRESS;
   function "+" (LEFT : INTEGER_TYPE;   RIGHT : SYSTEM.ADDRESS)
      return SYSTEM.ADDRESS;
   function "-" (LEFT : SYSTEM.ADDRESS; RIGHT : SYSTEM.ADDRESS)
      return INTEGER_TYPE;
   function "-" (LEFT : SYSTEM.ADDRESS; RIGHT : INTEGER_TYPE)
      return SYSTEM.ADDRESS;

   function TO_ADDRESS (X : INTEGER_TYPE) return ADDRESS;
   function TO_INTEGER_TYPE (X : ADDRESS) return INTEGER_TYPE;

private

   pragma INLINE ("+","-", TO_ADDRESS, TO_INTEGER_TYPE);

end ADDRESS_OPERATIONS;
pragma INLINE_GENERIC (ADDRESS_OPERATIONS);
```

Example:

```
X : INTEGER;
A : SYSTEM.ADDRESS := X'ADDRESS;     -- legal

function FETCH  is new FETCH_FROM_ADDRESS(INTEGER);
procedure ASSIGN is new ASSIGN_TO_ADDRESS(INTEGER);

X := FETCH(A);                         -- like "X := A.all;"
ASSIGN(A,X);                           -- like "A.all := X;"
```

This example assumes that the representations of the type INTEGER and the type ADDRESS are the same. On systems where the representations are different, the code needs to take that into account. For example, on Alpha systems, the type INTEGER is represented in 32 bits and the type ADDRESS is represented in 32 or 64 bits (32 bits on OpenVMS Alpha systems and 64 bits on Digital UNIX systems). On these systems, the example needs to be rewritten as follows:

```
type ADDRESS_INTEGER is
   -2**(SYSTEM.ADDRESS_SIZE -1) .. 2**(SYSTEM.ADDRESS_SIZE -1) - 1;
 . . .

X : ADDRESS_INTEGER;
A : SYSTEM.ADDRESS := X'ADDRESS;     -- legal

function FETCH  is new FETCH_FROM_ADDRESS(ADDRESS_INTEGER);
procedure ASSIGN is new ASSIGN_TO_ADDRESS(ADDRESS_INTEGER);

X := FETCH(A);                         -- like "X := A.all;"
ASSIGN(A,X);                           -- like "A.all := X;"
```

References: actual parameter 6.4 6.4.1, boolean predefined type 3.5.3, deferred constant 7.4, discriminant 3.3 3.7.1, erroneous 1.6, formal parameter 6.4, function 6.5, generic subprogram 12.1 12.1.3, instantiation 12.3, integer type 3.5.4, operation 3.3.3, private type 7.4 7.4.1, range 3.5, subtype 3.3, system.max_int 13.7, type 3.3, unconstrained array type 3.6

13.7a.2 Enumeration Type for Identifying Type Classes

DEC Ada declares the following enumeration type in the package SYSTEM for identifying the various Ada type classes:

```
type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                    TYPE_CLASS_INTEGER,
                    TYPE_CLASS_FIXED_POINT,
                    TYPE_CLASS_FLOATING_POINT,
                    TYPE_CLASS_ARRAY,
                    TYPE_CLASS_RECORD,
                    TYPE_CLASS_ACCESS,
                    TYPE_CLASS_TASK,
                    TYPE_CLASS_ADDRESS);
```

In addition to the usual operations for discrete types (see 3.5.5), DEC Ada provides the attribute TYPE_CLASS.

For every type or subtype T:

T′TYPE_CLASS   Yields the value of the type class for the full type of T. If T is a generic formal type, then the value is the value for the corresponding actual subtype. The value of this attribute is of the type TYPE_CLASS.

This attribute is only allowed within a unit to which the predefined package SYSTEM applies.

Examples:

Given

```
type MY_INT is range 1 .. 10;
type NEW_INT is new STRING;
package PACK is
   type PRIV is private;
private
   type PRIV is new FLOAT;
end PACK;
```

then

```
-- MY_INT'TYPE_CLASS    equals    TYPE_CLASS_INTEGER
-- NEW_INT'TYPE_CLASS   equals    TYPE_CLASS_ARRAY
-- PRIV'TYPE_CLASS      equals    TYPE_CLASS_FLOATING_POINT
```

References: access type 3.8, address type 13.7 13.7a.1, array type 3.6, discrete type 3.5, enumeration type 3.5.1, fixed point type 3.5.9, floating point type 3.5.7, generic formal type 12.1 12.1.2, integer type 3.5.4, record type 3.9, subtype 3.3 3.3.2, task type 9.1

13.7a.3 Non-Ada Exception

DEC Ada declares the following exception in the package SYSTEM to allow the treatment of non-Ada conditions or signals as a special subclass of Ada exceptions:

```
NON_ADA_ERROR : exception;
```

When used as a choice in an Ada exception part, the exception NON_ADA_ ERROR matches itself and other non-Ada exceptions as follows:

• On OpenVMS systems, it matches the following:

– Itself.

- Any OpenVMS condition whose facility field is not ADA and which does not match an Ada exception (see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information).

- Ada exceptions for which the pragma IMPORT_EXCEPTION or EXPORT_EXCEPTION is given and for which the VMS format has been specified.

- On Digital UNIX systems, it matches itself and any Digital UNIX signals that denote error conditions and are not already matched by predefined Ada exceptions (for example, signals that are matched by the exceptions CONSTRAINT_ERROR, STORAGE_ERROR, and so on).

See the DEC Ada run-time reference manuals for more information on the matching of Ada exceptions to system conditions or signals.

Example:

```
-- mixed-language example involving errors that can be handled
-- only by a handler for the exception NON_ADA_ERROR (or for others)
--
with TEXT_IO, SYSTEM; use TEXT_IO, SYSTEM;
procedure NON_ADA_EX is

    function CALL_C (INT_VAL : in INTEGER) return INTEGER;

    pragma INTERFACE (C, CALL_C);

begin

    -- call C routine, passing integer parameter
    --
    TEXT_IO.PUT (INTEGER'IMAGE (CALL_C(12)));

exception
    when NON_ADA_ERROR =>
        TEXT_IO.PUT_LINE ("NON_ADA_ERROR handled");
end NON_ADA_EX;

/*
 * C routine expects a pointer to an integer rather
 * than an integer; causes a system error
 */
call_c (int_ptr)
int *int_ptr;
{
    return *int_ptr * 2;
}
```

In this example, the mismatch in declarations for the INT_VAL and int_ptr parameters causes an access violation on OpenVMS systems (ACCVIO) and a segmentation violation (SIGSEGV) on Digital UNIX systems. These errors cannot be handled with handlers for any of the Ada predefined exceptions. They can be handled with a handler for the exception NON_ADA_ERROR (or with the more global **others** choice).

References: exception 11, exception declaration 11.1

13.7a.4 Hardware-Oriented Types and Functions

DEC Ada declares the following types, subtypes, and functions in the package SYSTEM for convenience in working with hardware-oriented storage:

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

subtype BIT_ARRAY_8  is BIT_ARRAY (0 ..  7);
subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

type UNSIGNED_BYTE is range 0 .. 255;
for  UNSIGNED_BYTE'SIZE use 8;

function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
function TO_BIT_ARRAY_8   (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

type UNSIGNED_WORD is range 0 .. 65535;
for  UNSIGNED_WORD'SIZE use 16;

function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
function TO_BIT_ARRAY_16  (X : UNSIGNED_WORD) return BIT_ARRAY_16;

type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

type UNSIGNED_LONGWORD is range -2_147_483_648 .. 2_147_483_647;
for  UNSIGNED_LONGWORD'SIZE use 32;

function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
```

```
function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32) return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

type UNSIGNED_LONGWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_LONGWORD;

-- On Alpha systems:
type UNSIGNED_32 is range 0 .. 4_294_967_295;
for  UNSIGNED_32'SIZE use 32;

-- On Alpha systems:
function "not" (LEFT        : UNSIGNED_32) return UNSIGNED_32;
function "and" (LEFT, RIGHT : UNSIGNED_32) return UNSIGNED_32;
function "or"  (LEFT, RIGHT : UNSIGNED_32) return UNSIGNED_32;
function "xor" (LEFT, RIGHT : UNSIGNED_32) return UNSIGNED_32;

-- On Alpha systems:
function TO_UNSIGNED_32 (X : BIT_ARRAY_32) return UNSIGNED_32;
function TO_BIT_ARRAY_32 (X : UNSIGNED_32) return BIT_ARRAY_32;

type UNSIGNED_QUADWORD is
   record
      L0 : UNSIGNED_LONGWORD;
      L1 : UNSIGNED_LONGWORD;
   end record;
for  UNSIGNED_QUADWORD'SIZE use 64;
for UNSIGNED_QUADWORD use
   record at mod 8;
   end record;

function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64) return UNSIGNED_QUADWORD;
function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;

type UNSIGNED_QUADWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_QUADWORD;
```

Note:

On Digital UNIX systems, these declarations are provided primarily for compatibility with Ada programs written for OpenVMS systems. on OpenVMS VAX systems, the type UNSIGNED_LONGWORD does not have an unsigned range. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

References: address predefined type 3.7, array 3.6, boolean predefined type 3.5.3, function 6.5, integer type 3.5.4, overloading 8.7, subtype 3.3 3.3.2, system.ast_handler type 13.7b.3, system.max_int 13.7, system.min_int 13.7, type 3.3, universal_integer type 3.5.4

13.7a.5 Convenient Unsigned Longword Declarations

The following DEC Ada package SYSTEM declarations provide convenient
names for static subtypes of the predefined type UNSIGNED_LONGWORD:

```
subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;
```

References: range 3.5, static subtype 4.9, subtype 3.3 3.3.2, system.unsigned_
longword 13.7a.4, type 3.3

13.7a.6 Global Symbol Values

The following DEC Ada function obtains the value of a link-time global symbol
(sometimes called a global literal):

```
function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;
```

On OpenVMS systems, if the parameter SYMBOL is a string literal of 31
or fewer characters, then the value of the function is the value of the global
symbol in the external environment.

On OpenVMS and Digital UNIX systems, the following DEC Ada functions also obtain a link-time global symbol as a value of the type ADDRESS or of the largest integer type:

```
function IMPORT_ADDRESS (SYMBOL : STRING)        return ADDRESS;

function IMPORT_LARGEST_VALUE (SYMBOL : STRING)  return LARGEST_INTEGER;
```

The maximum length for the string literal is related to the maximum size of a system address or integer (31 characters on OpenVMS systems; 63 characters on Digital UNIX systems).

Examples:

```
CMS_CREATED : constant CONDITION_HANDLING.COND_VALUE_TYPE
    := IMPORT_VALUE("CMS$_CREATED");
```

The preceding constant declaration is used in a DEC Ada program that calls DEC Code Management System (CMS) routines. The IMPORT_VALUE function is used to assign the value of the global symbol CMS$_CREATE to the condition value constant (an unsigned longword) CMS_CREATED, so that the success or failure of a CMS creation routine can be monitored. (The specification of the DEC Ada package CONDITION_HANDLING is presented in the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*.)

```
with SYSTEM, TEXT_IO;
use SYSTEM, TEXT_IO;
procedure IMPORT_VALUE_EXAMPLE is
    ETEXT : constant LARGEST_INTEGER := IMPORT_LARGEST_VALUE ("etext");
begin
    PUT_LINE ("Address of etext: " & LARGEST_INTEGER'IMAGE(ETEXT));
end IMPORT_VALUE_EXAMPLE;
```

When the preceding example is linked into an image on a Digital UNIX system, the Ada constant ETEXT is assigned the address of the symbol etext. The symbol is created by the linker, and its value indicates an address after the text segment of the image. The text segment is often or can be made read-only). See end(3) and a.out(5) for more information about etext and Digital UNIX image structure.

References: constant declaration 3.2, function 6.1, function result 6.5, parameter 6.2, string literal 2.6, string type 3.6.3

13.7b System-Specific DEC Ada Additions to the Package System

In addition to the language-required declarations in and the general DEC Ada additions to the package SYSTEM, DEC Ada declares the operations, constants, types, and subtypes described in the following sections. The information in these sections applies to individual DEC Ada compilers.

13.7b.1 VAX Floating Point Types (OpenVMS Systems Only)

DEC Ada declares four floating point types in the package SYSTEM. These types correspond to the four VAX hardware floating point data representations: D_floating, F_floating, G_floating, and H_floating.

```
type D_FLOAT is implementation_defined;
type F_FLOAT is implementation_defined;
type G_FLOAT is implementation_defined;
type H_FLOAT is implementation_defined; -- on VAX systems only
```

These types have all of the properties of floating types in general (operators, attributes, implicit conversion of real literals, and so on). See section 3.5.7 and the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for explanations of the VAX floating point types and type representations.

13.7b.2 IEEE Floating Point Types (OpenVMS Alpha and Digital UNIX Systems Only)

DEC Ada declares two floating point types in the package SYSTEM. These types correspond to the two IEEE floating point data representations: IEEE single float and IEEE double float.

```
type IEEE_SINGLE_FLOAT is
implementation_defined;
type IEEE_DOUBLE_FLOAT is
implementation_defined;
```

These types have all of the properties of floating types in general (operators, attributes, implicit conversion of real literals, and so on). See section 3.5.7 and the DEC Ada run-time reference manuals for explanations of the IEEE floating point types and type representations.

13.7b.3 OpenVMS Asynchronous-System-Trap-Related Declarations (OpenVMS Systems Only)

To support a means of handling OpenVMS asynchronous system traps (ASTs), and to allow the explicit specification of imported OpenVMS system service routines that involve ASTs, DEC Ada declares the following type and its auxiliary constant in the package SYSTEM:

```
type AST_HANDLER is limited private;

NO_AST_HANDLER : constant AST_HANDLER;
```

The type AST_HANDLER is used in the specification of the AST_ENTRY attribute, which is the only operation defined for this type. A value of the type AST_HANDLER is the address of a specially created routine that implements the delivery of an AST as a call to a particular entry in a particular task. See section 9.12a.

When used as an actual parameter to an imported OpenVMS system service routine, the constant NO_AST_HANDLER allows explicit specification that no AST handler is provided.

A function for converting values of the type AST_HANDLER to the type SYSTEM.UNSIGNED_LONGWORD is as follows:

```
function TO_UNSIGNED_LONGWORD (X : AST_HANDLER)
    return UNSIGNED_LONGWORD;
```

See section 13.7a.4 for information about the type SYSTEM.UNSIGNED_ LONGWORD.

Note:

When using the type AST_HANDLER to pass a parameter to an imported subprogram, it is generally necessary to specify the VALUE mechanism option. See section 13.9a.1.4 and the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*.

Example:

```
-- interface for the Get Job/Process Information system service
```

```
procedure GETJPI (
    STATUS : out COND_VALUE_TYPE;
    EFN    : in  EF_NUMBER_TYPE   := EF_NUMBER_ZERO;
    PIDADR : in  ADDRESS          := ADDRESS_ZERO;
    PRCNAM : in  PROCESS_NAME_TYPE:= PROCESS_NAME_TYPE'NULL_PARAMETER;
    ITMLST : in  ITEM_LIST_TYPE;
    IOSB   : out IOSB_TYPE;
    ASTADR : in  AST_HANDLER      := NO_AST_HANDLER;
    ASTPRM : in  USER_ARG_TYPE    := USER_ARG_ZERO);
pragma INTERFACE (EXTERNAL, GETJPI);
pragma IMPORT_VALUED_PROCEDURE (GETJPI, "SYS$GETJPI",
    (COND_VALUE_TYPE, EF_NUMBER_TYPE, ADDRESS,
     PROCESS_NAME_TYPE, ITEM_LIST_TYPE, IOSB_TYPE,
     AST_HANDLER, USER_ARG_TYPE),
    (VALUE, VALUE, VALUE, DESCRIPTOR(S), REFERENCE, REFERENCE,
     VALUE, VALUE));
```

As the preceding example shows, the type AST_HANDLER and constant NO_
AST_HANDLER are useful in writing interfaces to OpenVMS system routines.
This specification from the package STARLET shows the use of the type AST_
HANDLER to declare an AST address parameter and shows the use of the
constant NO_AST_HANDLER to give that parameter a default value.

References: actual parameter 6.4 6.4.1, asynchronous system trap 9.12a, attribute
4.1.4, constant 3.2.1, entry 9.5, exception 11, exception declaration 11.1, importing
subprograms 13.9a.1.1, limited private type 7.4.4, operation 3.3.3, task 9, type 3.3

13.7b.4 VAX Processor and Device Register Operations (OpenVMS VAX Systems Only)

The following package SYSTEM operations allow access to VAX processor
device registers:

```
function  READ_REGISTER (SOURCE : UNSIGNED_BYTE)
    return UNSIGNED_BYTE;
function  READ_REGISTER (SOURCE : UNSIGNED_WORD)
    return UNSIGNED_WORD;
function  READ_REGISTER (SOURCE : UNSIGNED_LONGWORD)
    return UNSIGNED_LONGWORD;

procedure WRITE_REGISTER(SOURCE : UNSIGNED_BYTE;
                         TARGET : out UNSIGNED_BYTE);
procedure WRITE_REGISTER(SOURCE : UNSIGNED_WORD;
                         TARGET : out UNSIGNED_WORD);
procedure WRITE_REGISTER(SOURCE : UNSIGNED_LONGWORD;
                         TARGET : out UNSIGNED_LONGWORD);
```

The READ_REGISTER functions return the value of a variable reference (byte,
word, or longword). The WRITE_REGISTER procedures load a specified value
or group of values into a specified target variable reference (byte, word, or
longword).

Each READ_REGISTER and WRITE_REGISTER operation is performed by a single machine instruction and is not affected by any compiler optimizations. Use of these operations is the only safe method for reading or writing a device register. These operations can also be used to read or write a variable in shared memory although use of the pragma SHARED is the preferred method of doing so.

References: exception 11, exception handling 11.4, function 6.1, integer type 3.5.4, procedure 6.1, system.unsigned_byte type 13.7a.4, system.unsigned_word type 13.7a.4, system.unsigned_longword type 13.7a.4, variable 3.2.1 3.7.3

13.7b.5 Interlocked-Instruction Procedures (OpenVMS and Digital UNIX Systems Only)

The following package SYSTEM procedures provide equivalents for VAX interlocked instructions:

```
-- On OpenVMS and Digital UNIX systems:
procedure CLEAR_INTERLOCKED (BIT       : in out BOOLEAN;
                             OLD_VALUE : out BOOLEAN);
procedure SET_INTERLOCKED   (BIT       : in out BOOLEAN;
                             OLD_VALUE : out BOOLEAN);

-- On OpenVMS and Digital UNIX systems:
type ALIGNED_WORD is
   record
      VALUE : SHORT_INTEGER;
   end record;
for  ALIGNED_WORD use
   record at mod 2;
   end record;

-- On Alpha systems only:
procedure CLEAR_INTERLOCKED (BIT          : in out BOOLEAN;
                             OLD_VALUE    : out BOOLEAN;
                             RETRY_COUNT  : in NATURAL;
                             SUCCESS_FLAG : out BOOLEAN);
procedure SET_INTERLOCKED   (BIT          : in out BOOLEAN;
                             OLD_VALUE    : out BOOLEAN;
                             RETRY_COUNT  : in NATURAL;
                             SUCCESS_FLAG : out BOOLEAN);

-- On OpenVMS and Digital UNIX systems:
procedure ADD_INTERLOCKED (ADDEND : in     SHORT_INTEGER;
                           AUGEND : in out ALIGNED_WORD;
                           SIGN   : out    INTEGER);
```

The procedure:

- CLEAR_INTERLOCKED is equivalent to the VAX Branch on Bit Clear and Clear Interlocked (BBCCI) instruction

- SET_INTERLOCKED is equivalent to the VAX Branch on Bit Set and Set Interlocked (BBSSI) instruction

- ADD_INTERLOCKED is equivalent to the VAX Add Aligned Word Interlocked (ADAWI) instruction.

These instructions interlock memory accesses and provide a means for synchronizing access to shared memory across processors.

The CLEAR_INTERLOCKED and SET_INTERLOCKED procedures clear or set a single bit and return the previous value of the bit.

The ADD_INTERLOCKED procedure adds two signed-word integers (ADDEND and AUGEND). SIGN is assigned the following integer result:

−1 if the new value of AUGEND is negative
0 if AUGEND is zero
+1 if AUGEND is positive

The type ALIGNED_WORD, used in the ADD_INTERLOCKED procedure, specifies a word-sized integer that is word-aligned (the type STANDARD.SHORT_INTEGER is a word-sized integer that is byte-aligned).

For more information on the use of these operations, see the DEC Ada run-time reference manuals..

References: boolean predefined type 3.5.3, integer type 3.5.4, procedure 6.1, type 3.3, short_integer predefined type 3.5.4

13.7b.6 Atomic Procedures (Alpha Systems Only)

The following package SYSTEM procedures provide atomic addition, AND, and OR operations for longwords and quadwords (4- and 8-byte quantities) on Alpha systems. These procedures guarantee that the operations performed will be indivisible.

```
-- On Alpha systems only:
type ALIGNED_INTEGER is
   record
      VALUE : INTEGER;
   end record;
for  ALIGNED_INTEGER use
   record at mod 4;
   end record;
```

```ada
-- On Alpha systems only:
type ALIGNED_LONG_INTEGER is
   record
      VALUE : LONG_INTEGER;
   end record;
for  ALIGNED_LONG_INTEGER use
   record at mod 8;
   end record;

-- For the following declarations, the declaration without
-- a RETRY_COUNT parameter mean to retry infinitely.  A value of 0
-- for the RETRY_COUNT parameter means do not retry.
--
-- On Alpha systems only:
procedure ADD_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      AMOUNT       : in INTEGER);
procedure ADD_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      AMOUNT       : in INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
procedure ADD_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      AMOUNT       : in LONG_INTEGER);
procedure ADD_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      AMOUNT       : in LONG_INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out LONG_INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);

-- On Alpha systems only:
procedure AND_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      FROM         : in INTEGER);
procedure AND_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      FROM         : in INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
procedure AND_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      FROM         : in LONG_INTEGER);
procedure AND_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      FROM         : in LONG_INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out LONG_INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
```

```
-- On Alpha systems only:
procedure OR_ATOMIC  (TO            : in out ALIGNED_INTEGER;
                      FROM          : in INTEGER);
procedure OR_ATOMIC  (TO            : in out ALIGNED_INTEGER;
                      FROM          : in INTEGER;
                      RETRY_COUNT   : in NATURAL;
                      OLD_VALUE     : out INTEGER;
                      SUCCESS_FLAG  : out BOOLEAN);
procedure OR_ATOMIC  (TO            : in out ALIGNED_LONG_INTEGER;
                      FROM          : in LONG_INTEGER);
procedure OR_ATOMIC  (TO            : in out ALIGNED_LONG_INTEGER;
                      FROM          : in LONG_INTEGER;
                      RETRY_COUNT   : in NATURAL;
                      OLD_VALUE     : out LONG_INTEGER;
                      SUCCESS_FLAG  : out BOOLEAN);
```

References: integer type 3.5.4, procedure 6.1, type 3.3

13.7b.7 Interlocked-Queue-Instruction Procedures (OpenVMS Systems Only)

The following package SYSTEM procedures provide equivalents for VAX interlocked queue instructions. Note that the types INSQ_STATUS and REMQ_STATUS are slightly different on VAX and Alpha systems.

```
-- On OpenVMS VAX systems only:
type INSQ_STATUS is (OK_NOT_FIRST, FAIL_NO_LOCK,
OK_FIRST);
for INSQ_STATUS use (OK_NOT_FIRST  => 0,
                     FAIL_NO_LOCK  => 1,
                     OK_FIRST      => 2);

-- On OpenVMS VAX systems only:
type REMQ_STATUS is (OK_NOT_EMPTY,  FAIL_NO_LOCK,
                     OK_EMPTY,      FAIL_WAS_EMPTY);
for REMQ_STATUS use (OK_NOT_EMPTY   => 0,
                     FAIL_NO_LOCK   => 1,
                     OK_EMPTY       => 2,
                     FAIL_WAS_EMPTY => 3);

-- On OpenVMS Alpha systems only:
type INSQ_STATUS is (FAIL_NO_LOCK, OK_NOT_FIRST,
OK_FIRST);
for INSQ_STATUS use (FAIL_NO_LOCK   => -1,
                     OK_NOT_FIRST   => 0,
                     OK_FIRST       => 1);
```

```
-- On OpenVMS Alpha systems only:
type REMQ_STATUS is (FAIL_NO_LOCK,  FAIL_WAS_EMPTY,
                     OK_NOT_EMPTY,  OK_EMPTY);
for REMQ_STATUS use (FAIL_NO_LOCK   => -1,
                     FAIL_WAS_EMPTY => 0,
                     OK_NOT_EMPTY   => 1,
                     OK_EMPTY       => 2);

-- On OpenVMS systems only:
procedure INSQHI (ITEM   : in  ADDRESS;
                  HEADER : in  ADDRESS;
                  STATUS : out INSQ_STATUS);

-- On OpenVMS systems only:
procedure REMQHI (HEADER : in  ADDRESS;
                  ITEM   : out ADDRESS;
                  STATUS : out REMQ_STATUS);

-- On OpenVMS systems only:
procedure INSQTI (ITEM   : in  ADDRESS;
                  HEADER : in  ADDRESS;
                  STATUS : out INSQ_STATUS);

-- On OpenVMS systems only:
procedure REMQTI (HEADER : in  ADDRESS;
                  ITEM   : out ADDRESS;
                  STATUS : out REMQ_STATUS);
```

The procedure:

- INSQHI is equivalent to the VAX Insert Entry into Queue at Head, Interlocked instruction

- INSQTI is equivalent to the VAX Insert Entry into Queue at Tail, Interlocked instruction

- REMQHI is equivalent to the VAX Remove Entry from Queue at Head, Interlocked instruction

- REMQTI is equivalent to the Remove Entry from Queue at Tail, Interlocked instruction

The types INSQ_STATUS and REMQ_STATUS are defined to represent the status results of the procedures for manipulating self-relative queues (queues where the forward and backward links are defined as offsets from one link to the next, rather than as virtual addresses).

The INSQHI, REMQHI, INSQTI, and REMQTI procedures perform queue insertion and removal operations at the head and tail of a self-relative queue. The address values of HEADER and ITEM must be quadword-aligned. The enumeration value assigned to STATUS gives the state of the queue after the operation has been executed.

The VAX INSQHI, REMQHI, INSQTI, and REMQTI instructions all have a parameter named "entry". Because "entry" is a reserved word in Ada, the "entry" parameter in each analogous DEC Ada procedure has been changed to ITEM.

For more information on the use of these operations, see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*.

References: address predefined type 13.7, boolean predefined type 3.5.3, integer type 3.5.4, parameter 6.2, procedure 6.1, record representation clause 13.4, record type 3.7, type 3.3, short_integer predefined type 3.5.4

13.8 Machine Code Insertions

1   A machine code insertion can be achieved by a call to a procedure whose sequence of statements contains code statements.

2   `code_statement ::= type_mark'`*record*`_aggregate;`

3   A code statement is only allowed in the sequence of statements of a procedure body. If a procedure body contains code statements, then within this procedure body the only allowed form of statement is a code statement (labeled or not), the only allowed declarative items are use clauses, and no exception handler is allowed (comments and pragmas are allowed as usual).

4   Each machine instruction appears as a record aggregate of a record type that defines the corresponding instruction. The base type of the type mark of a code statement must be declared within the predefined library package called MACHINE_CODE; this package must be named by a with clause that applies to the compilation unit in which the code statement occurs. An implementation is not required to provide such a package.

DEC Ada does not provide the package MACHINE_CODE; a user-provided package named MACHINE_CODE cannot be used for machine code insertions.

5   An implementation is allowed to impose further restrictions on the record aggregates allowed in code statements. For example, it may require that expressions contained in such aggregates be static expressions.

6   An implementation may provide machine-dependent pragmas specifying register conventions and calling conventions. Such pragmas must be documented in Appendix F.

DEC Ada does not provide machine-dependent pragmas specifying register conventions. Calling conventions are specified by the interface and import-export pragmas (see 13.9 and 13.9a).

7 **Example:**
```
M : MASK;
procedure SET_MASK; pragma INLINE(SET_MASK);

procedure SET_MASK is
   use MACHINE_CODE;
begin
   SI_FORMAT'(CODE => SSM, B => M'BASE_REG, D => M'DISP);
   --  M'BASE_REG and M'DISP are implementation-specific
   --  predefined attributes
end;
```

8 **References:** allow 1.6, apply 10.1.1, comment 2.7, compilation unit 10.1, declarative item 3.9, exception handler 11.2, inline pragma 6.3.2, labeled statement 5.1, library unit 10.1, package 7, pragma 2.8, procedure 6 6.1, procedure body 6.3, record aggregate 4.3.1, record type 3.7, sequence of statements 5.1, statement 5, static expression 4.9, use clause 8.4, with clause 10.1.1

13.9 Interface to Other Languages

1 A subprogram written in another language can be called from an Ada program provided that all communication is achieved via parameters and function results. A pragma of the form

2   **pragma** INTERFACE (*language*_name, *subprogram*_name);

3 must be given for each such subprogram; a subprogram name is allowed to stand for several overloaded subprograms. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. The pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. The pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram. A body is not allowed for such a subprogram (not even in the form of a body stub) since the instructions of the subprogram are written in another language.[21]

4 This capability need not be provided by all implementations. An implementation may place restrictions on the allowable forms and places of parameters and calls.

Use of this pragma in DEC Ada is equivalent to supplying the body of the named subprogram or subprograms. So, the following rules apply:

• If a subprogram body is given later for a subprogram named with a pragma INTERFACE, the body is illegal.

---
[21] See also Appendix G, AI-00180 and AI-00298.

- If a pragma INTERFACE names a subprogram body, the pragma is illegal.

- If a duplicate pragma INTERFACE is given, the latter pragma is illegal.

In DEC Ada, a name specified by a pragma INTERFACE can be declared by a renaming declaration. In this case, the pragma INTERFACE applies to the subprogram only if the following all occur in the same declarative part or package specification:

- The subprogram that has been renamed

- The renaming declaration

- The pragma

The pragma is ignored if these conditions are not satisfied.

In addition, DEC Ada interprets the effect of a pragma INTERFACE such that implicit declarations of subprograms (such as predefined operators, derived subprograms, attribute functions, and so on) are accepted and ignored.

Depending upon its use in a DEC Ada program, a pragma INTERFACE is interpreted in combination with one of the three DEC Ada import subprogram pragmas: IMPORT_FUNCTION, IMPORT_PROCEDURE, or IMPORT_ VALUED_PROCEDURE. These pragmas are described in section 13.9a.1.1.

If a pragma INTERFACE is used without one of these import pragmas, a default interpretation is used, as follows:

- If the subprogram name applies to a single subprogram, then a default import pragma is assumed.

  For a function, the default is:

  ```
  pragma IMPORT_FUNCTION (function_designator);
  ```

  For a procedure, the default is:

  ```
  pragma IMPORT_PROCEDURE (procedure_identifier);
  ```

- If the subprogram name applies to two or more subprograms, the pragma applies to all of them. However, a warning is given if the appropriate DEC Ada import pragmas are not given for all of the subprograms.

Whether or not the pragma INTERFACE is used with an import pragma, the following rules apply:

- The language name must be an identifier.

- The language name determines a set of default characteristics for the subprogram; for example, the parameter passing mechanisms, the return mechanism (if the subprogram is a function), and so on.

On OpenVMS systems, the language name can be ADA, BLISS, C, FORTRAN, or DEFAULT. Any other language name or identifier causes the set of DEFAULT defaults to be chosen.

On Digital UNIX systems, the language name can be ADA, BLISS, C, or FORTRAN. Any other language name or identifier causes the set of C defaults to be chosen.

For some parameter or result types, no default mechanism is chosen. In those cases, if an explicit mechanism is not specified, the pragma INTERFACE is ignored.

See the DEC Ada run-time reference manuals for more information about the defaults.

- The subprogram name must be either an identifier or a string literal that denotes an operator symbol.

5   **Example:**

```
package FORT_LIB is
    function SQRT(X : FLOAT) return FLOAT;
    function EXP (X : FLOAT) return FLOAT;
private
    pragma INTERFACE(FORTRAN, SQRT);
    pragma INTERFACE(FORTRAN, EXP);
end FORT_LIB;
```

In DEC Ada, the preceding example is interpreted as follows: the pragma INTERFACE specifies that the indicated routines SQRT and EXP are to be imported and used as bodies for the Ada functions SQRT and EXP in the package FORT_LIB.

```
package CHOOSE_R is
    procedure P(X : INTEGER);
    procedure P(X : FLOAT);
private
    procedure R(X : FLOAT) renames P;
    pragma INTERFACE(C, R);
end CHOOSE_R;
```

In this example, the preceding pragma INTERFACE indicates that the body for the second procedure P is to be imported as routine R.

Notes:

6 The conventions used by other language processors that call Ada programs are not part of the Ada language definition. Such conventions must be defined by these other language processors.

7 The pragma INTERFACE is not defined for generic subprograms.

The meaning of the subprogram name is determined as for any name (see 8.3), except that the name can denote more than one subprogram. In the following declaration, the pragma INTERFACE applies to the first two procedures; it does not apply to the third because the declaration is not visible at the place of the pragma. This same interpretation is made for pragmas used to import and export subprograms (see 13.9a.1).

```
procedure P (B: BOOLEAN);
procedure P (I: INTEGER);
pragma INTERFACE (NONADA, P);
procedure P (F: FLOAT);
```

If a pragma INTERFACE and a pragma INLINE are used together, the pragma INLINE is ignored regardless of the order in which the two pragmas appear.

8 **References:** allow 1.6, body stub 10.2, compilation unit 10.1, declaration 3.1, declarative item 3.9, declarative part 3.9, function result 6.5, library unit 10.1, must 1.6, name 4.1, overloaded subprogram 6.6, package specification 7.1, parameter of a subprogram 6.2, pragma 2.8, subprogram 6, subprogram body 6.3, subprogram call 6.4, subprogram declaration 6.1

attribute 4.1.4, declarative part 3.9, derived subprogram 3.4, function 6.5, identifier 2.3, operator 4.5, operator symbol 6.1, procedure 6.1, renaming declaration 8.5, string literal 2.6

13.9a DEC Ada Import and Export Pragmas

DEC Ada provides import and export pragmas designed specifically for constructing programs composed of both Ada and non-Ada entities. The *import* pragmas allow an Ada program to refer to entities written in another language. The *export* pragmas make Ada entities available to programs written in other languages. The names of the pragmas indicate the kind of entity involved, as follows:

• Pragmas applying to (nongeneric) functions:

    IMPORT_FUNCTION
    EXPORT_FUNCTION

• Pragmas applying to (nongeneric) procedures:

    IMPORT_PROCEDURE

IMPORT_VALUED_PROCEDURE
        EXPORT_PROCEDURE
        EXPORT_VALUED_PROCEDURE

- Pragmas applying to objects:

        IMPORT_OBJECT
        EXPORT_OBJECT
        COMMON_OBJECT

On OpenVMS systems, DEC Ada provides the following additional pragmas:

- The pragma PSECT_OBJECT applies to objects (and is equivalent to the pragma COMMON_OBJECT).

- The pragmas IMPORT_EXCEPTION and EXPORT_EXCEPTION apply to exceptions.

All of these pragmas are described in this section, summarized in Annex B, and listed in Appendix F.

DEC Ada also provides the import-export pragma INTERFACE_NAME. This pragma is described in section 13.9b, summarized in Annex B, and listed in Appendix F.

The form of all of the DEC Ada import and export pragmas is as follows:

```
pragma import_export_pragma_name
    (internal_name [, external_designator]
      [, pragma_specific_options]);

import_export_pragma_name ::=  COMMON_OBJECT
    | EXPORT_EXCEPTION     -- OpenVMS systems only
    | EXPORT_FUNCTION
    | EXPORT_OBJECT
    | EXPORT_PROCEDURE
    | EXPORT_VALUED_PROCEDURE
    | IMPORT_EXCEPTION     -- OpenVMS systems only
    | IMPORT_FUNCTION
    | IMPORT_OBJECT
    | IMPORT_PROCEDURE
    | IMPORT_VALUED_PROCEDURE
    | PSECT_OBJECT         -- OpenVMS systems only

internal_name ::= [INTERNAL =>] simple_name
    | [INTERNAL =>] operator_symbol -- EXPORT_FUNCTION and
                                    -- IMPORT_FUNCTION only

external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal
```

The internal name can be an Ada simple name, or, if the declared entity is a function, the internal name can be a string literal that denotes an operator symbol. A subprogram to be imported or exported must be uniquely identified by its internal name and parameter types and, in the case of a function, the result type (see 13.9a.1.1).

The external designator determines a symbol that is referenced or declared in the linker object module. If a string literal is given, the exact value (including the case of any letters) is used as the symbol. The value must be acceptable to the linker. If an identifier is given, the identifier is used to determine the symbol. If no external designator is given, the internal name is used to determine the symbol.

On OpenVMS systems, the external designator must be 31 or fewer characters. On Digital UNIX systems, the external designator must be less than or equal to 63 or fewer characters. If the external designator has more than the allowed number of characters, the import or export pragma is ignored.

Pragma-specific options are described in the individual pragma sections that follow.

The DEC Ada import and export pragmas are only allowed at the place of a declarative item and must apply to an entity declared by an earlier declarative item of the same declarative part or package specification. At most one import or export pragma is allowed for any given entity. In the case of multiple overloaded subprograms, this rule applies to each subprogram independently.

Additional placement and usage rules that apply to particular pragmas are described in the individual pragma sections that follow.

Note:

On OpenVMS systems, external designator string literals containing dollar signs ($) are reserved for identifiers of Digital-supplied software components.

Parameter associations for DEC Ada import and export pragmas may be either positional or named. With positional association, the parameters are interpreted in the same order as they appear in the syntax definition. The rules for the mixing of positional and named association are the same as those that apply to subprograms (see 6.4).

A pragma for an entity declared in a package specification must not be given in the package body. A pragma for an entity given in the visible part of a package specification can, however, be given in either the visible or private part of the specification.

DEC Ada does not check that exported symbols do not conflict with each other or with other global symbols. Such checking is performed by the linker.

References: allow 1.6, declaration 3.1, declarative item 3.9, declarative part 3.9, entity 3.1, exception 11, function 6.5, generic subprogram 12.1, identifier 2.3, named parameter association 6.4, object 3.2, operator symbol 4.5 6.1, overloading 6.6, package body 7.1, package specification 7.1, parameter 6.2, positional parameter association 6.4, pragma 2.8, private part 7.2, procedure 6.1, program 10.1, renaming declaration 8.5, result type 5.8, simple name 3.1 4.1, string literal 2.6, string type 3.6.3, subprogram 6, visibility 8.3

13.9a.1 Importing and Exporting Subprograms

DEC Ada provides a series of pragmas that make it possible to call (nongeneric) subprograms in a mixed-language programming environment. The pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_ PROCEDURE specify that the body of the subprogram associated with an Ada subprogram specification is to be provided from another programming language. The pragma INTERFACE must precede one of these import pragmas (see 13.9).

The pragmas EXPORT_FUNCTION, EXPORT_PROCEDURE, and EXPORT_ VALUED_PROCEDURE allow an Ada procedure or function to be called from another programming language.

References: function 6.5, generic subprogram 12.1, pragma 2.8, procedure 6.1, subprogram 6, subprogram body 6.3, subprogram specification 6.1

13.9a.1.1 Importing Subprograms

DEC Ada provides three pragmas for importing subprograms: IMPORT_ FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE. The pragma INTERFACE_NAME can also be used to import subprograms; see 13.9b.

The pragmas IMPORT_FUNCTION and IMPORT_PROCEDURE allow an Ada program to call external (non-Ada) functions and procedures.

The pragma IMPORT_VALUED_PROCEDURE allows an Ada program to call an external routine that (like a function) returns a result value but that (like a procedure) can also cause side effects on its parameters. (A function with **in out** or **out** parameters is not legal in Ada.)

The pragma IMPORT_VALUED_PROCEDURE allows a routine to be interpreted as a procedure in the environment of an Ada program and as a function in the external environment. The function result is returned in the first parameter. This pragma is especially useful for calling routines written in other programming languages that return a status value but that can also update their parameters.

The form of the pragmas for importing subprograms is as follows:

```
pragma IMPORT_FUNCTION
    | IMPORT_PROCEDURE | IMPORT_VALUED_PROCEDURE
    (internal_name [, external_designator]
        [, [PARAMETER_TYPES  =>] (parameter_types)]
        [, [RESULT_TYPE       =>] type_mark]  -- IMPORT_FUNCTION
                                              -- only
        [, [MECHANISM         =>] mechanism]
        [, [RESULT_MECHANISM =>] mechanism_name]-- IMPORT_FUNCTION
                                                -- only
        [, [FIRST_OPTIONAL_PARAMETER =>] identifier]); -- OpenVMS
                                                       -- systems
                                                       -- only

parameter_types ::= null | type_mark {, type_mark}

mechanism ::=
    mechanism_name | (mechanism_association {, mechanism_association })

mechanism_association ::=
    [formal_parameter =>] mechanism_name

mechanism_name ::=
      VALUE
    | REFERENCE
    | DESCRIPTOR [([CLASS =>] class-name)] -- OpenVMS
                                          -- systems only

class_name ::= UBS | UBSB | UBA | S | SB | A | NCA
```

Functions must be uniquely identified by their internal names and parameter and result types. The parameter and result types can be omitted only if there is exactly one function of that name in the same declarative part or package specification. Otherwise, both the parameter and result types must be specified.

Procedures must be uniquely identified by their internal names and parameter types. The parameter types can be omitted only if there is exactly one procedure of that name in the same declarative part or package specification. Otherwise, the parameter types must be specified.

While the internal name can denote a subprogram renaming declaration, the import pragma actually applies to the *base subprogram declaration*. The base subprogram is obtained by following any sequence of renaming declarations back to the first nonrenaming subprogram declaration. Except for the rules related to the internal name itself, the rules and requirements for the use of these pragmas are given in terms of the base subprogram.

The external designator denotes a linker global symbol that is associated with the external subprogram. The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- When the external designator is an identifier or when no external designator is given (the internal name is used), the compiler chooses a default spelling for the symbol as follows:

    - On OpenVMS systems, the symbol is spelled in uppercase letters

    - On Digital UNIX systems, the symbol is spelled in lowercase letters

    - On Digital UNIX systems, if the language Fortran is also specified in the associated pragma INTERFACE, the symbol is spelled in lowercase letters with an appended underscore

The parameter types option specifies a series of one or more type marks (type or subtype names), not parameter names. Each type mark is positionally associated with a formal parameter in the subprogram's declaration. The absence of parameters must be indicated by the reserved word **null**.

The result type option is used only for functions. It specifies the type or subtype of the function result.

The mechanism option specifies how the imported subprogram expects its parameters to be passed (for example, by value, by reference, or by descriptor). The calling Ada program is responsible for ensuring that parameters are passed in the form required by the external routine. Mechanism options and possible values for mechanism names and class names are described in section 13.9a.1.4.

If the first form of mechanism is given (a single mechanism name without parentheses), all parameters are passed using that mechanism. If the second form of mechanism is given (a series of mechanism names or associations given in parentheses and separated by commas), positional or named association can be used for each mechanism name.

The result mechanism option specifies how the imported function expects its result to be returned (for example, by value, by reference, or by descriptor). The calling program (namely, the Ada program) is responsible for ensuring that the function result is passed in the form required by the external routine. Possible values for mechanism names and class names are described in section 13.9a.1.4.

On OpenVMS systems, the first optional parameter specifies the name of the first parameter that can be omitted from the parameter list in a call to the imported subprogram. This option is designed to be used with subprograms that allow parameters to be omitted with a truncated argument list. It optimizes the code generated for calls to those subprograms.

This option is supported only on OpenVMS systems.

The first optional parameter must specify the name of a formal parameter of the base subprogram. That formal parameter and all following formal parameters, if any, of the base subprogram must be of mode **in** and must have been specified with default expressions.

When this option is specified, the parameter list generated for the call to the imported subprogram is truncated as follows:

- Beginning at the end of the parameter list and working back toward the beginning of the list, actual parameters are omitted—or truncated. This happens provided that the evaluation of each actual parameter has no side effects (assignments to variables or input-output actions) and provided that one of the following conditions is true:

  - The actual parameter is implicit. It has been omitted in the call to the subprogram, allowing the default expression to take effect.

  - The value of the actual parameter is equal to the value of the corresponding default expression. (This condition is only considered if the values of both the actual parameter and its default expression are known at compilation time.)

- Truncation stops when either an actual parameter cannot be omitted or when the actual parameter associated with the formal parameter specified as the first optional parameter has been omitted.

If either the actual parameter or the default expression is or contains a function call, then a determination of equality or of side effects cannot be made. In other cases, the determination of equality or the absence of side effects may depend on the specifics of the situation.

In addition to the rules given in section 13.9a, the following rules apply to all DEC Ada imported subprograms:

- If an import pragma is given for a subprogram specification, the pragma INTERFACE (see 13.9) must also be given for the subprogram earlier in the same declarative part or package specification. The use of the pragma INTERFACE requires that a corresponding body is not given.

- If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit.

- The procedure specification that corresponds to a pragma IMPORT_ VALUED_PROCEDURE must have at least one parameter, and the first (or only) parameter must be of mode **out**.

- These pragmas can be used for subprograms declared with a renaming declaration. The internal name must be a simple name. The given pragma applies to the named subprogram only if the subprogram that has been renamed, the renaming declaration, and the pragma all occur in that same declarative part or package specification. The pragma is ignored if these conditions are not satisfied.

- None of these pragmas can be used for a generic subprogram or a generic subprogram instantiation. They also cannot be used for a subprogram that is declared by an instantiation of a predefined generic subprogram (such as UNCHECKED_CONVERSION).

Examples:
```
function SQRT (X : FLOAT) return FLOAT;
pragma INTERFACE(C, SQRT);
pragma IMPORT_FUNCTION (SQRT, FSQRT, (FLOAT), FLOAT);
```

In the preceding example, the pragma INTERFACE identifies SQRT as an external subprogram. The language name argument C identifies the external subprogram as being written in the C language.

The pragma IMPORT_FUNCTION uses positional notation to specify arguments for importing the declared function SQRT. The pragma form indicates that the internal name is SQRT, and the external designator is FSQRT (on OpenVMS systems, the associated linker symbol will be spelled in uppercase; on Digital UNIX systems, in lowercase).

The parameter type is FLOAT, and the result is of the type FLOAT. Because no parameter or result passing mechanism is specified, default mechanisms will be used (in this case, for the C language).

```
function SQRT (X : LONG_FLOAT) return LONG_FLOAT;
pragma INTERFACE (C, SQRT);
pragma IMPORT_FUNCTION  (INTERNAL        => SQRT,
                         PARAMETER_TYPES => (LONG_FLOAT),
                         RESULT_TYPE     => LONG_FLOAT,
                         EXTERNAL        => SQRT);
```

The preceding example shows an alternative way of importing the declared function SQRT using named notation and different parameter and result types. If this example is combined with the code in the first example (with only one occurrence of the pragma INTERFACE), then the result is an overloading of SQRT, as follows:

```
function SQRT (X : FLOAT) return FLOAT;
function SQRT (X : LONG_FLOAT) return LONG_FLOAT;
pragma INTERFACE (C, SQRT);
pragma IMPORT_FUNCTION (SQRT, FSQRT, (FLOAT), FLOAT);
pragma IMPORT_FUNCTION (INTERNAL        => SQRT,
                        PARAMETER_TYPES => (LONG_FLOAT),
                        RESULT_TYPE     => LONG_FLOAT,
                        EXTERNAL        => SQRT);
```

In the preceding example, the parameter and result passing mechanisms are again not specified, forcing the use of default mechanisms (for the C language).

```
procedure CHANGE (X,Y : INTEGER);
procedure EXCHANGE (X,Y : INTEGER) renames CHANGE;
pragma INTERFACE (FORTRAN, EXCHANGE);
pragma IMPORT_PROCEDURE (INTERNAL => EXCHANGE,
                         PARAMETER_TYPES => (INTEGER,INTEGER));
```

The preceding example shows the use of renaming with an imported procedure. (It is assumed that these declarations occur in a declarative part or package specification.) The renaming causes the imported Fortran procedure to be used in calls to both of the procedures CHANGE and EXCHANGE. Because no external designator is specified, the internal name is used to spell the linker global symbol.

On OpenVMS systems, the spelling is "EXCHANGE". On Digital UNIX systems, the spelling is "exchange_".

References: actual parameter 6.2 6.4 6.4.1, allow 1.6, compilation unit 10.1, declarative part 3.9, external designator 13.9a.1, formal parameter 6.1 6.2, function 6.5, function call 6.4, function result 6.5, generic instantiation 12.3, generic subprogram 12.1, internal name 13.9a.1, name 4.1, null reserved word 2.9, overloading 6.6, package specification 7.1, parameter mode 6.2, positional parameter association 6.4, pragma 2.8, procedure 6.1, procedure specification 6.1, program 10.1, renaming declaration 8.5, reserved word 2.9, result type 5.8, simple identifier 4.1, static expression 4.9, subprogram 6, subprogram body 6.3, subprogram declaration 6.1, subprogram specification 6.1, subtype 3.3.2, type 3.3.1, type mark 3.3.2

13.9a.1.2 Attribute for Optional Parameters

The OpenVMS calling standard provides conventions for optional arguments in OpenVMS routines (any non-Ada routines) that differ substantially from the conventions for default parameters in Ada subprograms. In an Ada subprogram with default parameters, the values of the parameters (if specified or absent) are evaluated during the call to the subprogram.

In a non-Ada routine, the absence of a parameter is indicated by a "shortened" argument list and/or by an argument list entry containing the address zero (for arguments that are passed by reference or by descriptor).

In these cases, depending on the kind of routine involved, default actions may be taken by the routine. The DEC Ada attribute NULL_PARAMETER allows the OpenVMS conventions for optional arguments in non-Ada routines to be used in calls to imported subprograms.

NULL_PARAMETER is supported on all systems.

For any type or subtype T:

T'NULL_PARAMETER   Denotes an (imaginary) object of type or subtype T allocated at (machine) address zero. The attribute is allowed only as the default expression of a formal parameter or as an actual expression of a subprogram call; in either case, the subprogram must be imported.

The identity of the object is represented by the address zero in the argument list, independent of the passing mechanism (explicit or default).

Example:

```
procedure CRMPSC (
   STATUS: out COND_VALUE_TYPE;
   INADR:  in  ADDRESS_RANGE_TYPE
      := ADDRESS_RANGE_TYPE'NULL_PARAMETER;
   RETADR: in  ADDRESS            := ADDRESS_ZERO;
   ACMODE: in  ACCESS_MODE_TYPE   := ACCESS_MODE_ZERO;
   FLAGS:  in  UNSIGNED_LONGWORD  := 0;
   GSDNAM: in  SECTION_NAME_TYPE
      := SECTION_NAME_TYPE'NULL_PARAMETER;
   IDENT:  in  SECTION_ID_TYPE
      := SECTION_ID_TYPE'NULL_PARAMETER;
   RELPAG: in  UNSIGNED_LONGWORD  := 0;
   CHAN:   in  CHANNEL_TYPE       := CHANNEL_ZERO;
   PAGCNT: in  UNSIGNED_LONGWORD  := 0;
   VBN:    in  UNSIGNED_LONGWORD  := 0;
   PROT:   in  FILE_PROTECTION_TYPE := FILE_PROTECTION_ZERO;
   PFC:    in  UNSIGNED_LONGWORD  := 0);
pragma INTERFACE (EXTERNAL, CRMPSC);
pragma IMPORT_VALUED_PROCEDURE (
   INTERNAL => CRMPSC,
   EXTERNAL => "SYS$CRMPSC",
   PARAMETER_TYPES => (COND_VALUE_TYPE, ADDRESS_RANGE_TYPE, ADDRESS,
      ACCESS_MODE_TYPE, UNSIGNED_LONGWORD, SECTION_NAME_TYPE,
      SECTION_ID_TYPE, UNSIGNED_LONGWORD, CHANNEL_TYPE,
      UNSIGNED_LONGWORD, UNSIGNED_LONGWORD, FILE_PROTECTION_TYPE,
      UNSIGNED_LONGWORD),
   MECHANISM => (VALUE, REFERENCE, VALUE, VALUE, VALUE,
      DESCRIPTOR(S), REFERENCE, VALUE, VALUE, VALUE, VALUE,
      VALUE, VALUE));
```

This example is one of the interfaces declared in the DEC Ada package STARLET for the OpenVMS system service SYS$CRMPSC. Default values are specified for all of the parameters except for the STATUS parameter because all of the parameters to SYS$CRMPSC are optional.

Note the use of the DEC Ada attribute NULL_PARAMETER to give default values to the INADR, GSDNAM, and IDENT parameters. These parameters are passed by reference, descriptor, and reference, respectively. Because of their passing mechanisms, they cannot be given the zero default values (0, ADDRESS_ZERO, and so on) that can otherwise be given to parameters that are passed by value.

References: attribute 4.1.4, formal parameter 6.2, object 3.2 3.2.1, parameter 6.2, parameter passing 13.9a.1.4, subprogram 6, subprogram call 6.4, subtype 3.3, system.address_zero 13.7a.1, type 3.3

13.9a.1.3 Exporting Subprograms

DEC Ada provides three pragmas for exporting subprograms:

> EXPORT_FUNCTION
> EXPORT_PROCEDURE
> EXPORT_VALUED_PROCEDURE

All three export pragmas establish an external name for a subprogram and make the name available to the linker as a global symbol, so that the subprogram can be called by a non-Ada routine.

The pragmas EXPORT_FUNCTION and EXPORT_PROCEDURE allow the export of the kind of subprograms indicated. The pragma EXPORT_VALUED_ PROCEDURE allows an external (non-Ada) routine to call an Ada procedure that (like a function) returns a result value but that (like a procedure) can cause side effects on its parameters (a function with **in out** or **out** parameters is not legal in Ada). The pragma EXPORT_VALUED_PROCEDURE allows a procedure written in Ada to be interpreted as a function in the external environment; the function result is returned in the first parameter. The pragma is especially useful for writing Ada subprograms that are called by external routines, where the Ada subprogram involved is expected to return a status value and may update its parameters.

The form of the pragmas for exporting subprograms is as follows:

```
pragma EXPORT_FUNCTION
    | EXPORT_PROCEDURE | EXPORT_VALUED_PROCEDURE
    (internal_name [, external_designator]
       [, [PARAMETER_TYPES =>] (parameter_types)]
       [, [RESULT_TYPE     =>] type_mark]        -- EXPORT_FUNCTION
                                                 -- only
       [, [MECHANISM       =>] mechanism]
       [, [RESULT_MECHANISM =>] mechanism_name]); -- EXPORT_FUNCTION
                                                 -- only
```

Functions must be uniquely identified by their internal names and parameter and result types. The parameter and result types can be omitted only if there is exactly one function of that name in the same declarative part or package specification. Otherwise, both the parameter and result types must be specified.

Procedures must be uniquely identified by their internal names and parameter types. The parameter types can be omitted only if there is exactly one procedure of that name in the same declarative part or package specification. Otherwise, the parameter types must be specified.

The external designator denotes a linker global symbol that is associated with the external subprogram. The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- When the external designator is an identifier or when no external designator is given (the internal name is used), the compiler chooses a default spelling for the symbol as follows:

  - On OpenVMS systems, the symbol is spelled in uppercase letters

  - On Digital UNIX systems, the symbol is spelled in lowercase letters

The parameter types option specifies a series of one or more type marks (type or subtype names), not parameter names. Each type mark is positionally associated with a formal parameter in the subprogram's declaration. The absence of parameters must be indicated by the reserved word **null**.

The result type option is used only for functions. It specifies the type or subtype of the function result.

The mechanism option specifies how the exported subprogram expects its parameters to be passed (for example, by value, by reference, or by descriptor). The calling non-Ada routine is responsible for ensuring that parameters are passed in the form required by the Ada subprogram. Mechanism options and possible values for mechanism names and class names are described in sections 13.9a.1.1 and 13.9a.1.4.

If a single mechanism name is specified without parentheses, all parameters are passed using that mechanism. If a series of mechanism names or associations are given in parentheses and separated by commas, positional or named association can be used for each mechanism name.

The result mechanism option specifies how the imported function expects its result to be returned (for example, by value, by reference, or by descriptor). The calling non-Ada routine is responsible for ensuring that the function result is passed in the form required by the Ada function. Possible values for mechanism names and class names are described in section 13.9a.1.4.

In addition to the rules given in section 13.9a, the rules for exporting subprograms are as follows:

- An exported subprogram must be a library unit or must be declared in the outermost declarative part of a library package. This means pragmas for exporting subprograms are allowed only in the following cases:

  - For a subprogram specification or a subprogram body that is a library unit.

  - For a subprogram specification that is declared in the outermost declarations of a package specification or a package body that is a library unit.

  - For a subprogram body that is declared in the outermost declarations of a package body that is a library unit.

  Consequently, an export pragma for a subprogram body is allowed only if either the body does not have a corresponding specification or if the specification and body occur in the same declarative part.

  This set of rules implies that a pragma EXPORT_FUNCTION, EXPORT_PROCEDURE, or EXPORT_VALUED_PROCEDURE cannot be given for a generic library subprogram, nor can one be given for a subprogram declared in a generic library package. However, any of these pragmas can be given for a subprogram resulting from the instantiation of a generic subprogram, provided that the instantiation otherwise satisfies this set of rules.

- In the case of a subprogram declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit.

- The procedure specification that corresponds to a pragma EXPORT_VALUED_PROCEDURE must have at least one parameter, and the first (or only) parameter must be of mode **out**.

- None of these pragmas can be used for a subprogram that is declared with a renaming declaration.

- None of these pragmas can be used for a subprogram that is declared by an instantiation of a built-in library generic subprogram (such as UNCHECKED_CONVERSION).

Examples:

```
procedure PROC (X : INTEGER);
pragma EXPORT_PROCEDURE (PROC);
```

This example shows an export pragma that causes the Ada procedure PROC to be exported for use in a non-Ada routine. The name PROC will be declared as a linker global symbol. On OpenVMS systems, the symbol will be spelled in uppercase; on Digital UNIX systems, in lowercase.

```
procedure MULTIPLY (X : out INTEGER;
                    Y : in out INTEGER) is
begin
   X := 10*Y;
end;
pragma EXPORT_VALUED_PROCEDURE (
   INTERNAL => MULTIPLY,
   PARAMETER_TYPES => (INTEGER,INTEGER));
---------------------------------------------------------------------

PROGRAM Call_Ada (INPUT, OUTPUT);

VAR
  A : INTEGER;

FUNCTION Multiply (VAR T : INTEGER) : INTEGER; EXTERN;

BEGIN
  A := 1;
  A := Multiply(A);
END.
```

The preceding example shows an Ada procedure being used as a function in a Pascal program. Because of the use of pragma EXPORT_VALUED_PROCEDURE, the first parameter of the Ada procedure MULTIPLY is recognized as a function result in the Pascal program Call_Ada; the value of the Ada parameter X is assigned to the Pascal variable A. The preceding example runs on OpenVMS systems only.

```
with SYSTEM;
package RECORD_PKG is

   HI : STRING (1 .. 11) := "HELLO WORLD";

   type VAR_STR_TYPE is
      record
         ADDR : SYSTEM.ADDRESS;
         LEN  : SHORT_INTEGER;
      end record;
   pragma COMPONENT_ALIGNMENT (COMPONENT_SIZE, VAR_STR_TYPE);
```

```ada
      procedure ASSIGN_RECORD (REC : out VAR_STR_TYPE);
      pragma EXPORT_PROCEDURE (
          INTERNAL => ASSIGN_RECORD,
          EXTERNAL => "assign_record",
          PARAMETER_TYPES => (VAR_STR_TYPE),
          MECHANISM => (REFERENCE));

   end RECORD_PKG;

   package body RECORD_PKG is

      procedure ASSIGN_RECORD (REC : out VAR_STR_TYPE) is
         begin
            REC.LEN := HI'LENGTH;
            REC.ADDR := HI'ADDRESS;
         end;

   end RECORD_PKG;
```
```
----------------------------------------------------------------------
#include <ada_init.h>

extern void assign_record();
extern void assign_rec_elab();

main()
{
   long init_handle;

   struct S {
      char *addr;
      short len;
   } record_obj;

   /*
    * initialize the component
    */
   ada_init_component (assign_rec_elab, DA_EXCLUSIVE,
      BHVR_ERROR, &init_handle);

   if (init_handle == 0)
      {
      write (1, "Ada_init_component failed.\n", 23);
      };

   /*
    * call Ada procedure, passing it a structure parameter by reference
    */
   assign_record (&record_obj);

   printf ("%.*s\n", record_obj.len, record_obj.addr);

   /*
    * finalize the component
    */
   ada_complete_component(&init_handle);
}
```

The preceding example shows an Ada procedure being called by a C program. The C program calls the Ada procedure to assign values to the C object record_ obj. Then, the C program prints out the value of the object. This example runs on Digital UNIX systems only.

On Digital UNIX systems, the C program must initialize and finalize the exported Ada procedure. See the DEC Ada Digital UNIX run-time reference and program development manuals for more information about exporting Ada subprograms on Digital UNIX systems.

References: allow 1.6, compilation unit 10.1, declarative part 3.9, external designator 13.9a.1, formal parameter 6.1, function result 6.5, generic package 12, generic subprogram 12, instantiation 12.3, internal name 13.9a.1, library unit 10.1, package body 7.3, package specification 7.1, parameter 6.2, parameter name 6.1, pragma 2.8, procedure 6 6.1, renaming declaration 8.5, reserved word 2.9, result type 5.8, simple identifier 4.1, subprogram 6, subprogram body 6.3, subprogram declaration 6.1, subprogram specification 6.1, subtype name 3.3.2, type mark 3.3.2, type name 3.3.1

13.9a.1.4 Controlling the Passing Mechanisms for Parameters and Function Results

The import and export pragmas allow the option of specifying the passing mechanisms for parameters and function results of imported or exported subprograms. The three available DEC Ada mechanism names are VALUE, REFERENCE, and DESCRIPTOR. These names force the use of specific mechanisms, and they can be used to guarantee that the specified mechanisms are used.

As shown in sections 13.9a.1.1 and 13.9a.1.3, mechanism names have the following form:

```
mechanism_name ::=
    VALUE
  | REFERENCE
  | DESCRIPTOR [([CLASS =>] class-name)]   -- OpenVMS
                                           -- systems only
```

The names are briefly defined as follows. See the DEC Ada run-time reference manuals for detailed information on which DEC Ada types can be passed by each mechanism.

VALUE	Specifies that the value of the actual parameter or function result is passed in the argument list or returned.
REFERENCE	Specifies that the address of the value of the actual parameter or function result is passed in the argument list or returned.

DESCRIPTOR

On OpenVMS systems, specifies that the address of a descriptor is passed or returned.

The descriptor contains the address of the value of the actual parameter or function result, plus additional information about the parameter or function result. The descriptor may include a class, specified with the following form:

```
DESCRIPTOR[([CLASS =>] class_name)]
```

If the class name is omitted, DEC Ada supplies defaults based on the type.

The possible class names and their meanings follow. Note the following definitions:

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 array of a discrete type whose components occupy successive, unsigned bytes.

Ada descriptor class name	Used for
UBS	an unaligned bit string
UBSB	an unaligned bit string with arbitrary bounds
UBA	an unaligned bit array
S	a string; also a scalar or access type parameter
SB	a string with arbitrary bounds
A	a contiguous array
NCA	a noncontiguous array

For more information on descriptors, see the *OpenVMS Calling Standard*.

To determine the default mechanisms chosen by the DEC Ada compiler, see the compiler-generated compilation notes, as follows:

- On OpenVMS systems, the compilation notes are available when the /WARNINGS=COMPILATION_NOTES qualifier is in effect during a compilation. See *Developing Ada Programs on OpenVMS Systems* for more information.

- On Digital UNIX systems, the compilation notes are available during a compilation when an ADAERRFLAGS or ADALISFLAGS environment variable is defined with a string that includes the letter c. See *Developing Ada Programs on DEC OSF/1 Systems* for more information.

Examples:

```
procedure P (X : STRING);
pragma INTERFACE(C, P);
pragma IMPORT_PROCEDURE (P, MECHANISM => DESCRIPTOR (A));
```

In the preceding example, the arguments given for the pragma IMPORT_ PROCEDURE indicate that the DESCRIPTOR mechanism and the A descriptor class are to be used to pass the formal parameter X.

```
type BIT_STRING is array (1 .. 32) of BOOLEAN;
pragma PACK(BIT_STRING);
function F (Y : BIT_STRING) return INTEGER;
pragma INTERFACE (C, F);
pragma IMPORT_FUNCTION (INTERNAL => F,
                        MECHANISM => VALUE,
                        RESULT_MECHANISM => VALUE);
```

In the preceding example, the arguments given for the pragma IMPORT_ FUNCTION indicate that the VALUE mechanism is to be used to pass the formal parameter Y and the function result.

```
procedure ASSIGN_RECORD (REC : out VAR_STR_TYPE);
pragma EXPORT_PROCEDURE (
   INTERNAL => ASSIGN_RECORD,
   PARAMETER_TYPES => (VAR_STR_TYPE),
   MECHANISM => REFERENCE);
```

In the preceding example, the arguments given for the pragma EXPORT_ PROCEDURE indicate that the REFERENCE mechanism is to be used to pass the formal parameter VAR_STR_TYPE.

References: access type 3.8, actual parameter 6.4, address type 13.7 13.7a.1, argument 2.8, discrete type 3.5, formal parameter mode 6.2, full type 7.4.1, private type 7.4 7.4.1, record type 3.7, scalar type 3.5, static 4.9, type declaration 3.3.1

13.9a.2 Importing and Exporting Objects

DEC Ada provides three pragmas for importing and exporting objects:

IMPORT_OBJECT
EXPORT_OBJECT
COMMON_OBJECT

The pragma IMPORT_OBJECT references storage declared in an external (non-Ada) routine.

The pragma EXPORT_OBJECT allows an external routine to refer to the storage allocated for an Ada object.

The pragma COMMON_OBJECT allows both Ada and non-Ada programs to share the same block of storage. So, the pragma COMMON_OBJECT functions as both an import and an export pragma. (The pragma INTERFACE_NAME can also be used to import and export objects. See 13.9b.)

On OpenVMS systems, DEC Ada also provides the pragma PSECT_OBJECT, which is synonymous with the DEC Ada pragma COMMON_OBJECT and has the same effect.

In addition to the rules given in section 13.9a, the rules for importing and exporting objects are as follows:

- The size of the object to be imported or exported must be known at compile time.

- Import and export pragmas are not allowed for objects declared with a renaming declaration.

Notes:

Objects of private or limited private types cannot be imported or exported outside of the package that declares the (limited) private type. They can be imported or exported inside the body of the package where the type is declared (that is, where the full type is known).

The DEC Ada pragmas for importing or exporting objects can precede or follow a pragma VOLATILE for the same objects (see 9.11).

Address clauses are not allowed in combination with any of the DEC Ada pragmas for importing or exporting objects. If used in such cases, the pragma involved is ignored (see 13.5).

References: array subtype 3.6, component 3.3, discriminant 3.3, generic unit 12 12.1, index constraint 3.6, limited private type 7.4.4, object 3.2, object declaration 3.2.1, package body 7.3, package specification 7.1, pragma 2.8, private type 7.4 7.4.1, record type 3.7, renaming declaration 8.5, scalar type 3.5, simple record type 13.9a.1.4, static constraint 4.9, subcomponent 3.3, subtype 3.3 3.3.2, variable 3.2.1, variant part 3.7.3

13.9a.2.1 Importing Objects

The DEC Ada pragma IMPORT_OBJECT specifies that the storage allocated for the object (when the external routine is compiled) be made known to the calling Ada program by an externally defined linker global symbol.

The form of this pragma is as follows:

```
pragma IMPORT_OBJECT
    (internal_name [, external_designator]
        [, [SIZE =>] external_symbol]);
```

The internal name is the object identifier.

The external designator denotes a linker global symbol that is associated with the external object. The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- Where the external designator is an identifier or the internal name is used, the compiler chooses a default spelling for the symbol as follows:

    – On OpenVMS systems, the symbol is spelled in uppercase letters

    – On Digital UNIX systems, the symbol is spelled in lowercase letters

The size option specifies a linker global symbol that will be defined in the object module. On OpenVMS systems, the symbol is the value of an absolute global symbol that will equal the size in bytes of the imported object. On OpenVMS systems, the size option can be used to achieve some level of link-time consistency checking. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information. On Digital UNIX systems, the size option is accepted, but has no effect.

In addition to the rules given in sections 13.9a and 13.9a.2, the following rule applies to the pragma IMPORT_OBJECT:

- Because it is not created by an Ada elaboration, the object to be imported cannot be a constant or require implicit initialization. Specifically, this restriction means that the object to be imported:

    – Cannot be a constant.

    – Cannot be an access type (which has a default initial value of **null**).

    – Cannot be a record type that has discriminants (which are always initialized) or components with default initial expressions.

    – Cannot be an object of a task type.

Examples:

```
PID : INTEGER;
pragma IMPORT_OBJECT (INTERNAL => PID,
                      EXTERNAL => "PROCESS$ID",
                      SIZE     => PSIZE);
```

In the preceding example, the variable PID refers to the externally defined OpenVMS symbol PROCESS$ID. The symbol PSIZE represents the default size for the symbol PROCESS$ID. In this case, PSIZE is 4 because PID is declared to be of type INTEGER, for which the Ada default is 4 bytes.

```
-- Ada declaration
--
GLOBAL_VAR : INTEGER;
pragma IMPORT_OBJECT (INTERNAL => GLOBAL_VAR,
                      EXTERNAL => "global_var");
--------------------------------------------------
/*
 * C declaration
 */
extern int global_var;
```

In the preceding example, the variable GLOBAL_VAR is imported so that it can refer to an equivalent variable declared in another Digital UNIX language (for example, C). The external designator explicitly specifies a lowercase spelling for the linker global symbol. Lowercase is the expected spelling for linker global symbols on Digital UNIX systems.

References: access type 3.8, component 3.3 3.7, constant 3.2.1, default initial value 3.8, discriminant 3.7.1, elaboration 3.1, expression 4.4, external designator 13.9a.1, external symbol 13.9a.1, identifier 2.3, implicit initial value 3.2.1, internal name 13.9a.1, integer type 3.5.4, object 3.2, pragma 2.8, record type 3.7, string literal 2.6 4.2, task type 9.2

13.9a.2.2 Exporting Objects

The DEC Ada pragma EXPORT_OBJECT specifies that when the Ada program is compiled, the storage allocated for the object be made known to other non-Ada programs by a linker global symbol.

The form of this pragma is as follows:

```
pragma EXPORT_OBJECT
    (internal_name [, external_designator]
       [, [SIZE =>] external_symbol]);
```

The internal name is the object identifier.

The external designator denotes a linker global symbol that is associated with the external object. The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- In the cases where the external designator is an identifier or the internal name is used, the compiler chooses a default spelling for the symbol as follows:

  - On OpenVMS systems, the symbol is spelled in uppercase letters

  - On Digital UNIX systems, the symbol is spelled in lowercase letters

The size option specifies a linker global symbol that will be defined in the object module. On OpenVMS systems, the symbol is the value of an absolute global symbol that will equal the size in bytes of the imported object. On OpenVMS systems, the size option can be used to achieve some level of link-time consistency checking. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information. On Digital UNIX systems, the size option is accepted but has no effect.

In addition to the rules given in sections 13.9a and 13.9a.2, the following rules apply to the pragma EXPORT_OBJECT:

- The object to be exported must be a constant or a variable declared with an object declaration at the outermost level of a library package specification or body.

- The pragma EXPORT_OBJECT is not allowed in a generic unit.

Examples:

```
PID: INTEGER;
pragma EXPORT_OBJECT (INTERNAL => PID,
                      EXTERNAL => "PROCESS$ID",
                      SIZE     => PSIZE);
```

In the preceding example, the variable PID refers to the externally defined OpenVMS symbol PROCESS$ID. The symbol PSIZE represents the default size for the symbol PROCESS$ID. In this case, PSIZE is 4 because PID is declared to be of type INTEGER, for which the Ada default is 4 bytes.

```
ADA_VAL : INTEGER := 54_321;
pragma EXPORT_OBJECT (INTERNAL => ADA_VAL);
```

In the preceding example, the Ada variable ADA_VAL is exported so that it can be called by another language. Once it is exported, ADA_VAL can be referenced by a C routine that declares its C equivalent (ada_val) with the **extern** specifier.

13.9a.2.3 Importing and Exporting Objects with the Pragma Common_Object

The pragma COMMON_OBJECT enables shared use of variables that are stored in overlaid linker storage areas. For example, the pragma COMMON_OBJECT is useful for referring to Fortran or BASIC common blocks or external variables in PL/I, C, or Pascal.

The pragma COMMON_OBJECT allows only one object to be allocated in a particular storage area.

The form of this pragma is as follows:

```
pragma COMMON_OBJECT
    (internal_name [, external_designator]
        [, [SIZE =>] external_symbol]);
```

On OpenVMS systems, the pragma PSECT_OBJECT is synonymous with the pragma COMMON_OBJECT and has the same syntax. Both pragmas are accepted in programs compiled on OpenVMS systems, and both have the same effect.

The internal name must be an Ada identifier that denotes a variable. The external designator names the linker storage area and can be either an Ada identifier or a string denoting an OpenVMS or Digital UNIX name.

When a pragma COMMON_OBJECT is used for exporting an object, the external designator establishes the name of the linker storage area. When a pragma COMMON_OBJECT is used for importing an object, the external designator refers to an existing (external) linker storage area. The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- In the cases where the external designator is an identifier or the internal name is used, the compiler chooses a default spelling for the symbol as follows:

    - On OpenVMS systems, the symbol is spelled in uppercase letters

    - On Digital UNIX systems, the symbol is spelled in lowercase letters

The size option specifies a linker global symbol that will be defined in the object module. In OpenVMS systems, the symbol is the value of an absolute global symbol that will equal the size in bytes of the imported object. On OpenVMS systems, the size option can be used to achieve some level of link-time consistency checking. On Digital UNIX systems, the size option is accepted, but has no effect.

In addition to the rules given in sections 13.9a and 13.9a.2, the following rule applies to the pragmas COMMON_OBJECT and PSECT_OBJECT:

• Because it is not created by an Ada elaboration, an object specified with a pragma COMMON_OBJECT or PSECT_OBJECT cannot be a constant or require implicit initialization. Specifically, this restriction means that the object:

  – Cannot be a constant.

  – Cannot be an access type (which has a default initial value of **null**).

  – Cannot be a record type that has discriminants (which are always initialized) or components with default initial expressions.

  – Cannot be an object of a task type.

See the DEC Ada run-time reference manuals for more information on using the pragmas COMMON_OBJECT and PSECT_OBJECT.

Note:

Unlike other programming languages, DEC Ada allows only one object to be stored in a linker storage area. However, by using record objects, the effect of storing multiple objects in one linker storage area can be achieved. Each record component then corresponds to one external variable.

Examples:

```
type BLOCK is
   record
      X1, X2, X3 : FLOAT;
   end record;

XS : BLOCK;
pragma COMMON_OBJECT (XS);
```

The preceding example of the pragma COMMON_OBJECT shows the allocation of the record variable XS with three components (X1, X2, X3) of the type FLOAT in the linker storage area XS. The name of the linker storage area is assumed to be the internal name XS because no external designator is given.

The code in this example represents the same linker storage area as the following named Fortran common block:

```
COMMON /XS/ X1, X2, X3
```

As specified, this declaration works on OpenVMS systems. For this declaration to work on Digital UNIX systems, the external designator must be specified, giving the spelling of the linker storage area expected by Fortran on Digital UNIX systems (on Digital UNIX systems, Fortran expects lowercase spelling with an appended underscore: "xs_").

```
-- Ada declarations for another shared common block

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 (STORAGE_UNIT, NESTED);

type REC is
   record
      FIRST  : NESTED;
      SECOND : INTEGER;
   end record;
pragma COMPONENT_ALIGNMENT (STORAGE_UNIT, REC);

OBJ : REC;

pragma COMMON_OBJECT (OBJ, "comm_");
```

The preceding example of the pragma COMMON_OBJECT shows the allocation of the record variable OBJ in the linker storage area "comm_". The external designator is explicitly specified so that it matches the name of its Fortran equivalent and so that it follows the spelling conventions required by Fortran on Digital UNIX systems. (On Digital UNIX systems, Fortran expects lowercase spelling with an appended underscore.)

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

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

RECORD /STRUCT/     OBJ
```

```
COMMON /COMM/     OBJ
```

The code in the preceding example represents the same linker storage area as the following named Fortran common block (COMM).

References: component type 3.7, external designator 13.9a.1, external symbol 13.9a.1, identifier 2.3, internal name 13.9a.1, object 3.2, pragma 2.8, record type 3.7, string 3.6.3, variable 3.2.1

13.9a.3 Importing and Exporting Exceptions (OpenVMS Systems Only)

On OpenVMS systems, DEC Ada provides the pragmas IMPORT_EXCEPTION and EXPORT_EXCEPTION for importing and exporting exceptions. The pragma IMPORT_EXCEPTION allows non-Ada (especially OpenVMS) exceptions to be used in Ada programs. The pragma EXPORT_EXCEPTION allows Ada exceptions to be used by external modules.

For clarity in describing these pragmas, the terms *exception* and *Ada exception* are used to refer to Ada exceptions as described in chapter 11. The terms *condition* and *OpenVMS condition* are used to refer to OpenVMS conditions as defined in the *OpenVMS Calling Standard*.

The rules for importing and exporting exceptions are given in section 13.9a. Any additional rules that apply are described in the following sections. Import and export pragmas are not allowed for exceptions declared with a renaming declaration.

Note:

A pragma for an exception that is declared in a package specification is not allowed in the package body.

References: exception 11, exception declaration 11.1, package body 7.3, package specification 7.1, pragma 2.8, renaming declaration 8.5

13.9a.3.1 Importing Exceptions (OpenVMS Systems Only)

On OpenVMS systems, the DEC Ada pragma IMPORT_EXCEPTION allows OpenVMS conditions (for example, from OpenVMS system services or other OpenVMS languages) to be propagated to Ada programs as Ada exceptions. This pragma specifies that the exception associated with an exception declaration in an Ada program be defined externally (in non-Ada code).

The form of this pragma is as follows:

```
pragma IMPORT_EXCEPTION
    (internal_name [, external_designator]
        [, [FORM =>] ADA | VMS]
        [, [CODE =>] static_integer_expression]);
```

The internal name must be an Ada identifier that denotes a declared exception. The external designator denotes a OpenVMS Linker global symbol to be used to refer to the exception. If no external designator is given, the internal name is used as the global symbol.

The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- In the cases where the external designator is an identifier or the internal name is used, the compiler spells the symbol in uppercase letters by default.

The form option indicates whether an Ada exception or a OpenVMS condition is being imported. The form ADA must be specified if an Ada exception is being imported. The form VMS (the default) must be specified if a OpenVMS condition is being imported.

If the form ADA is specified, then the external designator refers to the address of an Ada exception name (the address of a counted ASCII string that names the exception). If the exception form VMS is specified, then the external designator refers to the value of a 32-bit condition value, which is determined according to OpenVMS conventions.

If the exception code option is specified, it is interpreted as the 32-bit value of a standard OpenVMS condition value. (The value must have a maximum size of 32 bits. For example, it should be in the range INTEGER'FIRST .. INTEGER'LAST).) This option is legal only if VMS is also specified for the form option either explicitly or by default. If an exception code option is also specified, then it is not legal to also specify an external designator. To do so would imply two definitions of the exception code value.

OpenVMS condition values are defined in the *OpenVMS Calling Standard*. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* explains the specification of code options in more detail.

Example:
```
ACCVIO : exception;
pragma IMPORT_EXCEPTION (ACCVIO, "SS$_ACCVIO");
```

In this example, the OpenVMS condition SS$_ACCVIO is imported as the global symbol SS$_ACCVIO. It is referred to within the Ada program by the internal name ACCVIO, and the value of the form option is the default (VMS).

References: exception 11, exception declaration 11.1, exception propagation 11.4.1 11.4.2, expression 4.4, external designator 13.9a.1, identifier 2.3, internal name 13.9a.1, pragma 2.8, system.max_int 13.7, system.min_int 13.7

13.9a.3.2 Exporting Exceptions (OpenVMS Systems Only)

On OpenVMS systems, the DEC Ada pragma EXPORT_EXCEPTION propagates Ada exceptions outside of the Ada program, so they can be handled by programs written in other OpenVMS languages. This pragma establishes an external name for an Ada exception and makes the name available to the OpenVMS Linker as a global symbol.

The form of this pragma is as follows:

```
pragma EXPORT_EXCEPTION
    (internal_name [, external_designator]
        [, [FORM =>] ADA | VMS]
        [, [CODE =>] static_integer_expression]);
```

The internal name must be an Ada identifier that denotes a declared exception. The external designator denotes a OpenVMS Linker global symbol to be used to refer to the exception. If no external designator is given, the internal name is used as the global symbol.

The external designator is interpreted as follows:

- When the external designator is a string literal, the exact value (including the case of any letters) is used as the symbol. If a null string is given, no global symbol is defined.

- When the external designator is an identifier or the internal name is used, the compiler spells the symbol in uppercase letters by default.

The form option indicates whether an Ada exception or a OpenVMS condition is being exported. The form ADA must be specified if an Ada exception is being exported. The form VMS, the default, must be specified if a OpenVMS condition is being exported.

If the form ADA is specified, then the external designator refers to the address of an Ada exception name (the address of a counted ASCII string that names the exception).

If the form VMS is specified, the code option must be specified. The value for the code option must have a size of 32 bits (be in the range INTEGER′FIRST .. INTEGER′LAST) and is interpreted as a standard OpenVMS condition value. The external designator denotes a OpenVMS Linker global symbol for the condition value. If no external designator is specified, the internal name is used as the global symbol.

OpenVMS condition values are defined in the *OpenVMS Calling Standard*. Then *DEC Ada Run-Time Reference Manual for OpenVMS Systems* explains the specification of code options in more detail.

In addition to the rules given in section 13.9a, the rules for exporting exceptions include the following:

- The EXPORT_EXCEPTION pragma is not allowed for exceptions that are declared in a generic unit.

Example:

```
ACCVIO : exception;
pragma EXPORT_EXCEPTION (ACCVIO, "MY_PACKAGE$_ACCVIO", ADA);
```

In this example, an Ada exception is exported as a global symbol.

References: exception 11, exception declaration 11.1, identifier 2.3, pragma 2.8, propagation of an exception 11.4.1 11.4.2, string 3.6.3, system.max_int 13.7, system.min_int 13.7

13.9b The Pragma Interface_Name

DEC Ada provides the pragma INTERFACE_NAME to associate an external symbol with the internal Ada name for a subprogram or object.

The form of this pragma is as follows:

```
pragma INTERFACE_NAME (internal_name, external_name);
```

```
internal_name ::= simple_name | operator_symbol
```

```
external_name ::= string_literal
```

The internal name can be an Ada simple name that denotes a subprogram or an object. If the declared entity is a function, the internal name can be a string literal that denotes an operator symbol.

While the internal name can denote a subprogram renaming declaration, the pragma INTERFACE_NAME actually applies to the base subprogram declaration. The base subprogram is obtained by following any sequence of renaming declarations back to the first nonrenaming subprogram declaration. Except for the rules related to the internal name itself, the rules and requirements for the use of this pragma are given in terms of the base subprogram.

The external name may be any string literal. The string literal is used as a linker global symbol that is associated with the external subprogram or object.

The pragma INTERFACE_NAME is only allowed at the place of a declarative item, and it must apply to an entity declared by an earlier declarative item of the same declarative part or package specification. This pragma can apply to more than one subprogram with the same internal name. In this case, the external name is used for all of the overloaded subprograms.

When the pragma INTERFACE_NAME applies to a subprogram, the following rules apply:

- If a pragma INTERFACE_NAME is given for a subprogram specification, the pragma INTERFACE (see 13.9) must also be given for the subprogram earlier in the same declarative part or package specification. The use of the pragma INTERFACE implies that a corresponding body is not given.

- A pragma IMPORT_PROCEDURE, IMPORT_FUNCTION, or IMPORT_VALUED_PROCEDURE cannot be used for the same subprogram named in the pragma INTERFACE_NAME. If this occurs, the second pragma is ignored.

- If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit.

- This pragma can be used for subprograms declared with a renaming declaration. The pragma INTERFACE_NAME applies to the named subprogram only if the subprogram that has been renamed, the renaming declaration, and the pragma all occur in that same declarative part or package specification. The pragma is ignored if these conditions are not satisfied.

- This pragma cannot be used for a generic subprogram or a generic subprogram instantiation. It also cannot be used for a subprogram that is declared by an instantiation of a predefined generic subprogram (such as the generic function UNCHECKED_CONVERSION).

When the pragma INTERFACE_NAME applies to an object, the following rules apply:

- The size of the object must be known at compile time.

- A pragma IMPORT_OBJECT, EXPORT_OBJECT, PSECT_OBJECT, or COMMON_OBJECT cannot be used for the same object named in the pragma INTERFACE_NAME. If this occurs, the second pragma is ignored.

- An address clause and a pragma INTERFACE_NAME cannot both apply to an object. If both are specified, the pragma INTERFACE_NAME is ignored.

- The pragma INTERFACE_NAME is not allowed in a generic unit.

- The object must not be declared with a renaming declaration.

- Because it is not created by an Ada elaboration, the object cannot be a constant or require implicit initialization. Specifically, this restriction means that the object:

  - Cannot be a constant.

  - Cannot be an access type (which has a default initial value of **null**).

  - Cannot be a record type that has discriminants (which are always initialized) or components with default initial expressions.

  - Cannot be an object of a task type.

Examples:
```
function SQRT (X : FLOAT) returns FLOAT;
pragma INTERFACE (C, SQRT);
pragma INTERFACE_NAME (SQRT, "SquareRoot");
```

In the preceding example, the pragma INTERFACE_NAME associates the Ada function SQRT with a C function named SquareRoot.

```
LATEST_VALUE : INTEGER;
pragma INTERFACE_NAME (LATEST_VALUE, "Latest_Value");
```

In the preceding example, the pragma INTERFACE_NAME associates the Ada object LATEST_VALUE with a variable that has the external name Latest_Value.

References: access type 3.8, address clause 13.5, body 3.9, body of a program unit 3.9, compilation unit 10.1, constant 3.2.1, declarative item 3.9, declarative part 3.9, discriminants 3.3 3.7.1, elaboration 3.1 3.9, function 6.1 6.5, generic instantiation 12.3, generic subprogram 12.1, generic unit 12 12.1, initialization 3.2.1, instantiation 12.3, null access value 3.8, object 3.2 3.2.1, operator symbol 6.1, overloading subprogram 6.6, package specification 7.1, pragma 2.8, record type 3.7, renaming declaration 8.5, simple name 4.1, string literal 2.6 4.2, subprogram 6, subprogram declaration 6.1, subprogram name 6.1, subprogram specification 6.1, task type 9.1 9.2

13.10 Unchecked Programming

1 The predefined generic library subprograms UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION are used for unchecked storage deallocation and for unchecked type conversions.

2   ```
     generic
         type OBJECT is limited private;
         type NAME   is access OBJECT;
     procedure UNCHECKED_DEALLOCATION(X : in out NAME);
     ```

3    ```
     generic
         type SOURCE is limited private;
         type TARGET is limited private;
     function UNCHECKED_CONVERSION(S : SOURCE) return TARGET;
     ```

4   **References:** generic subprogram 12.1, library unit 10.1, type 3.3

13.10.1 Unchecked Storage Deallocation

1   Unchecked storage deallocation of an object designated by a value of an access type is achieved by a call of a procedure that is obtained by instantiation of the generic procedure UNCHECKED_DEALLOCATION. For example: [22]

     ```
     procedure FREE is new UNCHECKED_DEALLOCATION(object_type_name,
                                                  access_type_name);
     ```

2   Such a FREE procedure has the following effect:

3   (a)   after executing FREE(X), the value of X is **null**;

4   (b)   FREE(X), when X is already equal to **null**, has no effect;

5   (c)   FREE(X), when X is not equal to **null**, is an indication that the object designated by X is no longer required, and that the storage it occupies is to be reclaimed.

6   If X and Y designate the same object, then accessing this object through Y is erroneous if this access is performed (or attempted) after the call FREE(X); the effect of each such access is not defined by the language.

     Notes:

7   It is a consequence of the visibility rules that the generic procedure UNCHECKED_DEALLOCATION is not visible in a compilation unit unless this generic procedure is mentioned by a with clause that applies to the compilation unit. [23]

8   If X designates a task object, the call FREE(X) has no effect on the task designated by the value of this task object. The same holds for any subcomponent of the object designated by X, if this subcomponent is a task object.

---

[22] See also Appendix G, AI-00355.

[23] See also Appendix G, AI-00356.

References: access type 3.8, apply 10.1.1, compilation unit 10.1, designate 3.8 9.1, erroneous 1.6, generic instantiation 12.3, generic procedure 12.1, generic unit 12, library unit 10.1, null access value 3.8, object 3.2, procedure 6, procedure call 6.4, subcomponent 3.3, task 9, task object 9.2, visibility 8.3, with clause 10.1.1

13.10.2 Unchecked Type Conversions

1 An unchecked type conversion can be achieved by a call of a function that is obtained by instantiation of the generic function UNCHECKED_ CONVERSION. [24]

2 The effect of an unchecked conversion is to return the (uninterpreted) parameter value as a value of the target type, that is, the bit pattern defining the source value is returned unchanged as the bit pattern defining a value of the target type. An implementation may place restrictions on unchecked conversions, for example, restrictions depending on the respective sizes of objects of the source and target type. Such restrictions must be documented in Appendix F.

DEC Ada supports the generic function UNCHECKED_CONVERSION with the following restrictions on the class of types involved:

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type.

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained type with discriminants.

Further, when the target type is a type with discriminants, the value resulting from a call of the conversion function resulting from an instantiation of UNCHECKED_CONVERSION is checked to ensure that the discriminants satisfy the constraints of the actual subtype.

If the size of the source value is greater than the size of the target subtype, then the high order bits of the value are ignored (truncated). If the size of the source value is less than the size of the target subtype, then the value is extended with zero bits to form the result value.

3 Whenever unchecked conversions are used, it is the programmer's responsibility to ensure that these conversions maintain the properties that are guaranteed by the language for objects of the target type. Programs that violate these properties by means of unchecked conversions are erroneous.

---

[24] See also Appendix G, AI-00355.

Note:

4 It is a consequence of the visibility rules that the generic function UNCHECKED_ CONVERSION is not visible in a compilation unit unless this generic function is mentioned by a with clause that applies to the compilation unit.

5 **References:** apply 10.1.1, compilation unit 10.1, erroneous 1.6, generic function 12.1, instantiation 12.3, parameter of a subprogram 6.2, type 3.3, with clause 10.1.1

actual subtype 12.3, discriminant 3.3 3.7.1, formal type 12.1, unconstrained array type 3.6, unconstrained type 3.3

14

Input-Output

1   Input-output is provided in the language by means of predefined packages.
The generic packages SEQUENTIAL_IO and DIRECT_IO define input-output
operations applicable to files containing elements of a given type. Additional
operations for text input-output are supplied in the package TEXT_IO. The
package IO_EXCEPTIONS defines the exceptions needed by the above three
packages. Finally, a package LOW_LEVEL_IO is provided for direct control of
peripheral devices.

In addition to the packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO,
DEC Ada provides packages for handling files containing elements of mixed
types. On all systems, DEC Ada provides a total of five predefined input-output
packages:

- SEQUENTIAL_IO

- DIRECT_IO

- TEXT_IO

- SEQUENTIAL_MIXED_IO

- DIRECT_MIXED_IO

On OpenVMS systems, DEC Ada provides four additional packages for relative
and indexed access to files:

- RELATIVE_IO

- INDEXED_IO

- RELATIVE_MIXED_IO

- INDEXED_MIXED_IO

On all systems, DEC Ada provides the package IO_EXCEPTIONS.
On OpenVMS systems, DEC Ada provides the additional package
AUX_IO_EXCEPTIONS, which defines the exceptions needed by the
relative and indexed input-output packages.

DEC Ada does not provide the package LOW_LEVEL_IO for general use; however, a DEC Ada package LOW_LEVEL_IO exists for the implementation of the other input-output packages.

Complete descriptions and specifications for the DEC Ada predefined input-output and exceptions packages appear in the sections that follow. Detailed information on DEC Ada input-output processing can be found in the DEC Ada run-time reference manuals.

2 **References** : direct_io package 14.2 14.2.4, io_exceptions package 14.5, low_level_io package 14.6, sequential_io package 14.2 14.2.2, text_io package 14.3

aux_io_exceptions package 14.4 14.5a, direct access 14.2, direct_mixed_io package 14.2 14.2b 14.2b.5, element 14.1a, indexed access 14.2a, indexed_io package 14.2a 14.2a.4, indexed_mixed_io package 14.2a 14.2b 14.2b.9, library package 10.1, mixed-type file 14.2b, relative access 14.2a, relative_io package 14.2a 14.2a.2, relative_mixed_io package 14.2a 14.2b 14.2b.7, sequential access 14.2, sequential_mixed_io package 14.2 14.2b 14.2b.3

14.1 External Files and File Objects

1 Values input from the external environment of the program, or output to the environment, are considered to occupy *external files.* An external file can be anything external to the program that can produce a value to be read or receive a value to be written. An external file is identified by a string (the *name*). A second string (the *form*) gives further system-dependent characteristics that may be associated with the file, such as the physical organization or access rights. The conventions governing the interpretation of such strings must be documented in Appendix F. [1]

The DEC Ada interpretation of form strings is also described in section 14.1b.

2 Input and output operations are expressed as operations on objects of some *file type*, rather than directly in terms of the external files. In the remainder of this chapter, the term *file* is always used to refer to a file object; the term *external file* is used otherwise. The values transferred for a given file must all be of one type.

On OpenVMS systems, DEC Ada uses OpenVMS Record Management Services (RMS) to perform operations on external files. On Digital UNIX systems, DEC Ada uses system calls to perform operations on external files.

The attributes of the external file, the storage medium, and the input-output package determine which operations can be used to manipulate data.

---

[1] See also Appendix G, AI-00355.

3   Input-output for sequential files of values of a single element type is defined by
     means of the generic package SEQUENTIAL_IO. The skeleton of this package
     is given below.

4   ```
     with IO_EXCEPTIONS;
     generic
        type ELEMENT_TYPE is private;
     package SEQUENTIAL_IO is
        type FILE_TYPE is limited private;
        type FILE_MODE is (IN_FILE, OUT_FILE);
        ...
        procedure OPEN (FILE : in out FILE_TYPE; ...);
        ...
        procedure READ (FILE : in FILE_TYPE;
                        ITEM : out ELEMENT_TYPE);
        procedure WRITE(FILE : in FILE_TYPE;
                        ITEM : in ELEMENT_TYPE);
        ...
     end SEQUENTIAL_IO;
     ```

5    In order to define sequential input-output for a given element type, an
     instantiation of this generic unit, with the given type as actual parameter,
     must be declared.  The resulting package contains the declaration of a file
     type (called FILE_TYPE) for files of such elements, as well as the operations
     applicable to these files, such as the OPEN, READ, and WRITE procedures.

5    Input-output for direct access files is likewise defined by a generic package
     called DIRECT_IO. Input-output in human-readable form is defined by the
     (nongeneric) package TEXT_IO.

     In DEC Ada, input-output for sequential access files is also defined by the
     (nongeneric) package SEQUENTIAL_MIXED_IO. Input-output for direct access
     files is defined by the (nongeneric) package DIRECT_MIXED_IO.

     Input-output for relative and indexed access to files is available on
     OpenVMS systems.  Relative access to files is defined by the generic package
     RELATIVE_IO and the (nongeneric) package RELATIVE_MIXED_IO. Indexed
     access to files is defined by the generic package INDEXED_IO and the
     (nongeneric) package INDEXED_MIXED_IO.

6    Before input or output operations can be performed on a file, the file must first
     be associated with an external file. While such an association is in effect, the
     file is said to be *open*, and otherwise the file is said to be *closed.*

7    The language does not define what happens to external files after the
     completion of the main program (in particular, if corresponding files have not

been closed).  The effect of input-output for access types is implementation-dependent. [2]

In DEC Ada, input-output for access or task types is erroneous.

8    An open file has a *current mode*, which is a value of one of the enumeration types

```
type FILE_MODE is
    (IN_FILE, INOUT_FILE, OUT_FILE);  --  for DIRECT_IO

type FILE_MODE is
    (IN_FILE, OUT_FILE);               --  for SEQUENTIAL_IO
                                       --  and TEXT_IO
```

9   These values correspond respectively to the cases where only reading, both reading and writing, or only writing are to be performed. The mode of a file can be changed.

For the additional DEC Ada input-output packages, an open file's current mode can be one of the following values:

```
type FILE_MODE is
    (IN_FILE, INOUT_FILE, OUT_FILE); -- on all systems for
                                     -- DIRECT_MIXED_IO; also on
                                     -- OpenVMS systems for
                                     -- RELATIVE_IO,
                                     -- RELATIVE_MIXED_IO,
                                     -- INDEXED_IO, and
                                     -- INDEXED_MIXED_IO
type FILE_MODE is
    (IN_FILE, OUT_FILE);             -- for SEQUENTIAL_MIXED_IO
```

10   Several file management operations are common to the three input-output packages. These operations are described in section 14.2.1 for sequential and direct files. Any additional effects concerning text input-output are described in section 14.3.1.

The file management operations for the additional DEC Ada input-output packages are essentially the same as those described in section 14.2.1 for sequential and direct files. Differences and additional effects are described in section 14.2a.1 for relative and indexed input-output and in section 14.2b.1 for mixed input-output.

---

[2]   See also Appendix G, AI-00466.

11   The exceptions that can be raised by a call of an input-output subprogram are
all defined in the package IO_EXCEPTIONS; the situations in which they can
be raised are described, either following the description of the subprogram
(and in section 14.4), or in Appendix F in the case of error situations that are
implementation-dependent. [3]

On OpenVMS systems, the DEC Ada packages RELATIVE_IO, INDEXED_IO,
RELATIVE_MIXED_IO, and INDEXED_MIXED_IO also use the additional
package AUX_IO_EXCEPTIONS. See section 14.5a for information on these
auxiliary exceptions.

Notes:

12   Each instantiation of the generic packages SEQUENTIAL_IO and DIRECT_IO
declares a different type FILE_TYPE; in the case of TEXT_IO, the type FILE_
TYPE is unique.

In DEC Ada each instantiation of the generic packages RELATIVE_IO and
INDEXED_IO declares a different type FILE_TYPE; in the case of the
packages SEQUENTIAL_MIXED_IO, DIRECT_MIXED_IO, RELATIVE_
MIXED_IO, and INDEXED_MIXED_IO, the type FILE_TYPE is declared
by the (nongeneric) package.

13   A bidirectional device can often be modeled as two sequential files associated
with the device, one of mode IN_FILE, and one of mode OUT_FILE. An
implementation may restrict the number of files that may be associated with a
given external file. The effect of sharing an external file in this way by several
file objects is implementation-dependent. [4]

On OpenVMS systems, DEC Ada permits the sharing of external files, and,
in addition, uses the RMS automatic locking facility to provide record-locking
capabilities. This means that when an external file is opened in more than
one place, the operations are automatically coordinated. Record locking
ensures that one task cannot add, delete, or modify a record of an external
file that is concurrently being accessed by another task. Record locking
also coordinates accesses between programs that are executing as separate
OpenVMS processes. This includes processes executing on different nodes of a
VAXcluster, VMScluster (including dual-architecture VMSclusters) or DECnet
network. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*
for more information on file sharing and record locking.

On Digital UNIX systems, DEC Ada has no special sharing or locking
capabilities. External files can be shared or locked at the operating-system
level, using Digital UNIX system calls.

---

[3]   See also Appendix G, AI-00279.
[4]   See also Appendix G, AI-00320.

14.1a File Elements

In DEC Ada, input-output operations for all nontext files are defined in terms of file elements. Values *retrieved* for an element of a file object are read from an element of the external file. Values *transferred* to an element of a file object are written to an element of the external file. On OpenVMS systems, each DEC Ada external file element corresponds to an RMS record.

For files containing values of mixed types, an element can represent a single value (just as an element in a file of uniform-type values represents a single value), or it can represent a set of values, or items. A mixed-type file, then, can be a file of elements of different types, or it can be a file of elements whose items have different types. DEC Ada provides an additional set of input-output operations, defined in terms of items, for mixed-type files. Section 14.2b explains these operations in more detail.

Input-output operations for text files are not defined in terms of elements. Rather, they are defined in terms of lines and in terms of values, or items, of various types. Section 14.3 explains text files and their operations in more detail.

14.1b Specification of the FORM Parameter in DEC Ada

All of the DEC Ada input-output packages provide CREATE and OPEN procedures that have a FORM parameter, which corresponds to the language-required form string (see 14.1). The FORM parameter determines the system-dependent characteristics or attributes associated with an external file when it is opened or created.

References: create procedure 14.2.1, external file 14.1, open procedure 14.2.1

14.1b.1 The FORM Parameter on OpenVMS Systems

On OpenVMS systems, the value of the FORM parameter can be one of the following:

- A string of statements of the OpenVMS Record Management Services (RMS) File Definition Language (FDL)

- A string referring to a text file of FDL statements (called an FDL file)

FDL is a special-purpose OpenVMS language for writing file specifications. These specifications are then used by DEC Ada run-time routines to create or open files.

See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for the rules governing the FORM parameter and for a general description of FDL.

See the *Guide to OpenVMS File Applications* and the *OpenVMS Record Management Utilities Reference Manual* for complete information on FDL.

On OpenVMS systems, each input-output package has a default string of FDL statements that is used to open or create a file. In general, specification of a FORM parameter is not necessary. It is never necessary in an OPEN procedure although it may be necessary in a CREATE procedure. The packages for which a value for the FORM parameter must be specified in a CREATE procedure are as follows:

- The packages DIRECT_IO and RELATIVE_IO require that a maximum element (record) size be specified in the FORM parameter if the item with which the package is instantiated is unconstrained.

- The packages DIRECT_MIXED_IO and RELATIVE_MIXED_IO require that a maximum element (record) size be specified in the FORM parameter.

- The packages INDEXED_IO and INDEXED_MIXED_IO require that information about keys be specified in the FORM parameter.

Any explicit FORM specification supersedes the default attributes of the governing input-output package. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* describes the default external file attributes of each input-output package.

References: create procedure 14.2.1, direct_io package 14.2 14.2.4, direct_mixed_io package 14.2 14.2b 14.2b.5, external file 14.1, indexed_io package 14.2a 14.2a.4, indexed_mixed_io package 14.2a 14.2b 14.2b.9, instantiation 12.3, item 14.2b, key 14.2a, open file 14.1, open procedure 14.2.1, relative_io package 14.2a 14.2a.2, relative_ mixed_io package 14.2a 14.2b 14.2b.7, string 2.6 3.6.3 4.2, text file 14.3, unconstrained type 3.3 3.3.2

14.1b.2 The FORM Parameter on Digital UNIX Systems

On Digital UNIX systems, the value of the FORM parameter conforms to the description of the FORM parameter in IEEE Standard 1003.5-1992, *IEEE Standard for Information Technology, POSIX Ada Language Interfaces*. The value of the FORM parameter must be a character string, defined as follows:

```
form_image ::=
   [field_name_identifier => field_value
      {,field_name_identifier => field _value}]

field_value ::= field_value_identifier | integer
```

The field names and field values supported by DEC Ada are as follows:

OWNER, GROUP, OTHER Determines the file permissions associated with the file.

The field value can be NONE, READ, WRITE, EXECUTE, or any combination of the latter three values separated by underscores (for example, READ_WRITE). The field values set the access permissions for the created file. If no field value is specified, the default value is READ_WRITE_EXECUTE.

The file permissions field names and values can be used only in the FORM parameter of a CREATE procedure. The exception USE_ ERROR is raised if they are used in the FORM parameter of an OPEN procedure.

APPEND Determines whether or not data can be appended to the file.

The field value must be either TRUE or FALSE. A value of TRUE causes any output to be written to the end of the named external file. If no field value is specified, the default value is FALSE.

The append field name and value can be used only in the FORM parameter of an OPEN procedure. The exception USE_ERROR is raised if they are used in the FORM parameter of a CREATE procedure.

BLOCKING
Determines whether or not blocking input-output is in effect.

The field value must be TASKS. This value causes a calling task (but not other tasks) to wait for the completion of any input-output operation on the file.

TERMINAL_INPUT
Specifies how characters are read from the keyboard. This field applies to text files only.

The field value must be either LINES or CHARACTERS. The value LINES causes the canonical terminal input. The value CHARACTERS causes noncanonical terminal input. If no field value is specified, the default value is LINES.

The terminal input field name and value have no effect if the file is not opened for input or if the file is not opened on a terminal.

FILE_STRUCTURE
Determines whether the file is first-in first-out (FIFO) or not.

The field value must be either REGULAR or FIFO. If no field value is specified, the default value is REGULAR.

The file structure field name and values can be used only in the FORM parameter of a CREATE procedure. The exception USE_ERROR is raised if they are used in the FORM parameter of an OPEN procedure. The exception USE_ERROR is also raised if

the field name and values are applied to files created or opened with operations from the packages DIRECT_IO and DIRECT_MIXED_IO.

FILE_DESCRIPTOR Specifies a Digital UNIX file descriptor for the Ada file being opened.

The field value specifies the file descriptor. The file descriptor must be open.

If the file descriptor is not open, if the file descriptor refers to an Ada file that is already open, or if the file descriptor refers to an Ada file with an incompatible mode, then the exception USE_ERROR is raised. The file descriptor option can be used only in the FORM parameter of an OPEN procedure.

PAGE_TERMINATORS Determines the treatment of line, page, and file terminators. This field applies to text files only.

The field value must be either TRUE or FALSE. A value of TRUE causes the external representation of line, page, and file terminators to be as defined in the DEC Ada run-time reference manuals. A value of FALSE causes the external file to have no page terminators. If no field value is specified, the default value is TRUE.

Output to the external file occurs as follows when the field value is FALSE: line terminators are represented by the character ASCII.LF, page terminators are omitted, and file terminators are represented by the physical end of the file. The exception USE_ERROR is raised when an explicit call is made to the procedure TEXT_IO.NEW_PAGE or an explicit call is made to the procedure TEXT_IO.SET_LINE and the current line number exceeds the value specified by the TO parameter.

	Input to the external file occurs as follows when the field value is FALSE: any occurrence of the character ASCII.FF is interpreted as the character ASCII.FF, not as a page terminator.
BUFFER_SIZE	Determines the size of the buffer used during file operations.
	The field value must be an integer. It specifies the number of bytes in the buffer.
ELEMENT_SIZE	Determines the maximum element size for a direct file.
	The field value must be an integer. It specifies the maximum number of bytes in the element.

Each input-output package has an implementation-defined value form string that is used to open or create a file. In general, specification of a FORM parameter is not necessary. The packages for which a value for the FORM parameter must be specified in a CREATE procedure are as follows:

• The package DIRECT_IO requires that a maximum element size be specified in the FORM parameter if the item with which the package is instantiated is unconstrained.

• The package DIRECT_MIXED_IO requires that a maximum element size be specified in the FORM parameter.

References: ascii predefined package C, canonical order 11.6, create procedure 14.2.1, direct_io package 14.2 14.2.4, direct_mixed_io package 14.2 14.2b 14.2b.5, element 14.1a, external file 14.1, instantiation 12.3, item 14.2b, open file 14.1, open procedure 14.2.1, string 2.6 3.6.3 4.2, task 9, unconstrained type 3.3 3.3.2

14.2 Sequential and Direct Files

1   Two kinds of access to external files are defined: *sequential access* and *direct access.* The corresponding file types and the associated operations are provided by the generic packages SEQUENTIAL_IO and DIRECT_IO. A file object to be used for sequential access is called a *sequential file*, and one to be used for direct access is called a *direct file.*

In DEC Ada, sequential and direct access are also provided in the predefined (nongeneric) packages SEQUENTIAL_MIXED_IO and DIRECT_MIXED_IO, which allow values of different types to be mixed in a file. The operations provided by these packages are described in section 14.2b.

2   For sequential access, the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or by the environment). When the file is opened, transfer starts from the beginning of the file.

3   For direct access, the file is viewed as a set of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position. The position of an element is specified by its *index*, which is a number, greater than zero, of the implementation-defined integer type COUNT. The first element, if any, has index one; the index of the last element, if any, is called the *current size*; the current size is zero if there are no elements. The current size is a property of the external file.

In DEC Ada, the integer type COUNT is the **range** 0 .. INTEGER ˈ LAST.

4   An open direct file has a *current index*, which is the index that will be used by the next read or write operation. When a direct file is opened, the current index is set to one. The current index of a direct file is a property of a file object, not of an external file.

5   All three file modes are allowed for direct files. The only allowed modes for sequential files are the modes IN_FILE and OUT_FILE.

6   **References** : count type 14.3, file mode 14.1, in_file 14.1, out_file 14.1

direct_io package 14.2 14.2.4, direct_mixed_io package 14.2 14.2b 14.2b.5, generic package 12.1, integer type 3.5.4, mixed_type file 14.1a, sequential_io package 14.2 14.2.2, sequential_mixed_io package 14.2 14.2b 14.2b.3

14.2.1 File Management

1   The procedures and functions described in this section provide for the control of external files; their declarations are repeated in each of the three packages for sequential, direct, and text input-output. For text input-output, the procedures CREATE, OPEN, and RESET have additional effects described in section 14.3.1.

The declarations of the procedures and functions described in this section are also repeated in each of the DEC Ada packages for relative, indexed, and mixed-type input-output. For relative input-output, the procedure CREATE has additional effects described in section 14.2a.1. For indexed input-output the procedures CREATE and RESET have additional effects described in section 14.2a.1. For direct, relative, and indexed mixed input-output, the procedure CREATE has additional effects described in section 14.2b.1.

2   **procedure** CREATE(FILE : **in out** FILE_TYPE;
                 MODE : **in** FILE_MODE := *default_mode*;
                 NAME : **in** STRING := "";
                 FORM : **in** STRING := "");

3   Establishes a new external file, with the given name and form, and
             associates this external file with the given file. The given file is left
             open. The current mode of the given file is set to the given access mode.
             The default access mode is the mode OUT_FILE for sequential and text
             input-output; it is the mode INOUT_FILE for direct input-output.
             For direct access, the size of the created file is implementation-
             dependent. A null string for NAME specifies an external file that is not
             accessible after the completion of the main program (a temporary file).
             A null string for FORM specifies the use of the default options of the
             implementation for the external file. [5]

             For direct files in DEC Ada, the size of a file that has just been created
             is zero.

             On OpenVMS systems, a name can be specified in FORM. However,
             NAME overrides any name given in FORM. If NAME is null, any
             name given in FORM is ignored, and a temporary file is created that
             is not accessible after the file is closed.[6] See the *DEC Ada Run-Time
             Reference Manual for OpenVMS Systems* for more information on
             naming external files on OpenVMS systems.

             On Digital UNIX systems, a file descriptor can be specified in FORM. If
             the external file referred to by the file descriptor is already open, then
             the Ada file is opened, and the file pointer is set to the beginning of the
             file.

4   The exception STATUS_ERROR is raised if the given file is already
             open. The exception NAME_ERROR is raised if the string given
             as NAME does not allow the identification of an external file. The
             exception USE_ERROR is raised if, for the specified mode, the
             environment does not support creation of an external file with the
             given name (in the absence of NAME_ERROR) and form. [7]

             In DEC Ada, the exception USE_ERROR is raised if the mode is IN_
             FILE. The exception USE_ERROR is also raised if any file attributes
             specified in FORM are not supported by the package.

---

[5] See also Appendix G, AI-00046 and AI-00247.
[6] See also Appendix G, AI-00046.
[7] See also Appendix G, AI-00332.

5   **procedure** OPEN(FILE : **in out** FILE_TYPE;
                      MODE : **in** FILE_MODE;
                      NAME : **in** STRING;
                      FORM : **in** STRING := "");

6   Associates the given file with an existing external file having the given
           name and form, and sets the current mode of the given file to the given
           mode. The given file is left open.

7   The exception STATUS_ERROR is raised if the given file is already
           open. The exception NAME_ERROR is raised if the string given as
           NAME does not allow the identification of an external file; in particular,
           this exception is raised if no external file with the given name exists.
           The exception USE_ERROR is raised if, for the specified mode, the
           environment does not support opening for an external file with the
           given name (in the absence of NAME_ERROR) and form. [8]

8   **procedure** CLOSE(FILE : **in out** FILE_TYPE);

9   Severs the association between the given file and its associated external
           file. The given file is left closed. [9]

10   The exception STATUS_ERROR is raised if the given file is not open.

11   **procedure** DELETE(FILE : **in out** FILE_TYPE);

12   Deletes the external file associated with the given file. The given file is
           closed, and the external file ceases to exist.

13   The exception STATUS_ERROR is raised if the given file is not open.
           The exception USE_ERROR is raised if (as fully defined in Appendix F)
           deletion of the external file is not supported by the environment.

           In DEC Ada on Digital UNIX systems, the exception USE_ERROR
           is raised if an attempt is made to delete an Ada file that was opened
           by specifying the file descriptor in the FORM parameter of an OPEN
           procedure.

14   **procedure** RESET(FILE : **in out** FILE_TYPE; MODE : **in** FILE_MODE);
     procedure RESET(FILE : **in out** FILE_TYPE);

15   Resets the given file so that reading from or writing to its elements
           can be restarted from the beginning of the file; in particular, for direct
           access this means that the current index is set to one. If a MODE
           parameter is supplied, the current mode of the given file is set to the
           given mode. [10]

---

[8]   See also Appendix G, AI-00332.
[9]   See also Appendix G, AI-00357.
[10] See also Appendix G, AI-00357.

16   The exception STATUS_ERROR is raised if the file is not open. The
         exception USE_ERROR is raised if the environment does not support
         resetting for the external file and, also, if the environment does not
         support resetting to the specified mode for the external file.

17   **function** MODE(FILE : **in** FILE_TYPE) **return** FILE_MODE;

18   Returns the current mode of the given file.

19   The exception STATUS_ERROR is raised if the file is not open.

20   **function** NAME(FILE : **in** FILE_TYPE) **return** STRING;

21   Returns a string which uniquely identifies the external file currently
         associated with the given file (and may thus be used in an OPEN
         operation). If an environment allows alternative specifications of the
         name (for example, abbreviations), the string returned by the function
         should correspond to a full specification of the name.

22   The exception STATUS_ERROR is raised if the given file is not open.[11]

         In DEC Ada on Digital UNIX systems, the exception USE_ERROR is
         raised if an attempt is made to obtain the name of an Ada file that was
         opened by specifying the file descriptor in the FORM parameter of an
         OPEN or CREATE procedure.

23   **function** FORM(FILE : **in** FILE_TYPE) **return** STRING;

24   Returns the form string for the external file currently associated with
         the given file. If an environment allows alternative specifications of
         the form (for example, abbreviations using default options), the string
         returned by the function should correspond to a full specification (that
         is, it should indicate explicitly all options selected, including default
         options).

         See section 14.1b for an explanation of the form string on DEC Ada
         systems.

         On OpenVMS systems, a full FDL string is returned. See the *DEC Ada
         Run-Time Reference Manual for OpenVMS Systems* for an explanation
         of FDL strings.

---

[11] See also Appendix G, AI-00046.

25   The exception STATUS_ERROR is raised if the given file is not open.

26 **function** IS_OPEN(FILE : **in** FILE_TYPE) **return** BOOLEAN;

27   Returns TRUE if the file is open (that is, if it is associated with an external file), otherwise returns FALSE.

28 **References** : current mode 14.1, current size 14.1, closed file 14.1, direct access 14.2, external file 14.1, file 14.1, file_mode type 14.1, file_type type 14.1, form string 14.1, inout_file 14.2.4, mode 14.1, name string 14.1, name_error exception 14.4, open file 14.1, out_file 14.1, status_error exception 14.4, use_error exception 14.4

in_file 14.1 string 3.6.3

14.2.2 Sequential Input-Output

1   The operations available for sequential input and output are described in this section. [12] The exception STATUS_ERROR is raised if any of these operations is attempted for a file that is not open.

See section 14.2.1 for descriptions of the file management operations that are available for sequential input and output.

2   **procedure** READ(FILE : **in** FILE_TYPE; ITEM : **out** ELEMENT_TYPE);

3   Operates on a file of mode IN_FILE. Reads an element from the given file, and returns the value of this element in the ITEM parameter.

4   The exception MODE_ERROR is raised if the mode is not IN_FILE. The exception END_ERROR is raised if no more elements can be read from the given file. The exception DATA_ERROR is raised if the element read cannot be interpreted as a value of the type ELEMENT_TYPE; however, an implementation is allowed to omit this check if performing the check is too complex.

In DEC Ada, this procedure does not perform the check that raises the exception DATA_ERROR.

5   **procedure** WRITE(FILE : **in** FILE_TYPE; ITEM : **in** ELEMENT_TYPE);

6   Operates on a file of mode OUT_FILE. Writes the value of ITEM to the given file.

7   The exception MODE_ERROR is raised if the mode is not OUT_FILE. The exception USE_ERROR is raised if the capacity of the external file is exceeded.

---

[12] See also Appendix G, AI-00320.

8 **function** END_OF_FILE(FILE : **in** FILE_TYPE) **return** BOOLEAN;

9   Operates on a file of mode IN_FILE. Returns TRUE if no more elements can be read from the given file; otherwise returns FALSE.

10   The exception MODE_ERROR is raised if the mode is not IN_FILE.

11 **References** : data_error exception 14.4, element 14.1, element_type 14.1, end_error exception 14.4, external file 14.1, file 14.1, file mode 14.1, file_type 14.1, in_file 14.1, mode_error exception 14.4, out_file 14.1, status_error exception 14.4, use_error exception 14.4

14.2.3 Specification of the Package Sequential_IO

```
1   with IO_EXCEPTIONS;
    generic
       type ELEMENT_TYPE is private;
    package SEQUENTIAL_IO is

       type FILE_TYPE is limited private;

       type FILE_MODE is (IN_FILE, OUT_FILE);

       -- File management

       procedure CREATE(FILE : in out FILE_TYPE;
                        MODE : in FILE_MODE := OUT_FILE;
                        NAME : in STRING := "";
                        FORM : in STRING := "");

       procedure OPEN  (FILE : in out FILE_TYPE;
                        MODE : in FILE_MODE;
                        NAME : in STRING;
                        FORM : in STRING := "");

       procedure CLOSE (FILE : in out FILE_TYPE);
       procedure DELETE(FILE : in out FILE_TYPE);
       procedure RESET (FILE : in out FILE_TYPE;
                        MODE : in FILE_MODE);
       procedure RESET (FILE : in out FILE_TYPE);

       function MODE    (FILE : in FILE_TYPE) return FILE_MODE;
       function NAME    (FILE : in FILE_TYPE) return STRING;
       function FORM    (FILE : in FILE_TYPE) return STRING;

       function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

       -- Input and output operations

       procedure READ  (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE);
       procedure WRITE (FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

       function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

       -- Exceptions
```

```
        STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
        MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
        NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
        USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
        DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
        END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
        DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

    private
        -- implementation-dependent
    end SEQUENTIAL_IO;
```

2   **References** : close procedure 14.2.1, create procedure 14.2.1, data_error exception 14.4, delete procedure 14.2.1, device_error exception 14.4, end_error exception 14.4, end_of_file function 14.2.2, file_mode 14.1, file_type 14.1, form function 14.2.1, in_ file 14.1, io_exceptions 14.4, is_open function 14.2.1, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_error exception 14.4, open procedure 14.2.1, out_file 14.1, read procedure 14.2.2, reset procedure 14.2.1, sequential_io package 14.2 14.2.2, status_error exception 14.4, use_error exception 14.4, write procedure 14.2.2,

14.2.4 Direct Input-Output

1   The operations available for direct input and output are described in this section. [13] The exception STATUS_ERROR is raised if any of these operations is attempted for a file that is not open.

See section 14.2.1 for descriptions of the file management operations that are available for direct input and output.

2   ```
     procedure READ(FILE : in FILE_TYPE;
                    ITEM : out ELEMENT_TYPE;
                    FROM : in POSITIVE_COUNT);

     procedure READ(FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE);
     ```

3   Operates on a file of mode IN_FILE or INOUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter FROM. Then (for both forms) returns, in the parameter ITEM, the value of the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

4   The exception MODE_ERROR is raised if the mode of the given file is OUT_FILE. The exception END_ERROR is raised if the index to be used exceeds the size of the external file. The exception DATA_ERROR is raised if the element read cannot be interpreted as a value of the type ELEMENT_TYPE; however, an implementation is allowed to omit this check if performing the check is too complex.

---

[13] See also Appendix G, AI-00320.

In DEC Ada, this procedure does not perform the check that raises the exception DATA_ERROR.

5
```
procedure WRITE(FILE : in FILE_TYPE;
                ITEM : in ELEMENT_TYPE;
                TO   : in POSITIVE_COUNT);
```

```
procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);
```

6   Operates on a file of mode INOUT_FILE or OUT_FILE. In the case of the first form, sets the index of the given file to the index value given by the parameter TO. Then (for both forms) gives the value of the parameter ITEM to the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

7   The exception MODE_ERROR is raised if the mode of the given file is IN_FILE. The exception USE_ERROR is raised if the capacity of the external file is exceeded.

8
```
procedure SET_INDEX(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
```

9   Operates on a file of any mode. Sets the current index of the given file to the given index value (which may exceed the current size of the file).

10
```
function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
```

11   Operates on a file of any mode. Returns the current index of the given file.

12
```
function SIZE(FILE : in FILE_TYPE) return COUNT;
```

13   Operates on a file of any mode. Returns the current size of the external file that is associated with the given file.

Returns the number of elements in the file (in DEC Ada). This value is obtained from the external file when an existing file is opened. The value is updated whenever the index of a WRITE operation to the file exceeds the current size. Therefore, the size of the file is always equal to the highest index number that has been written to the file.

On OpenVMS systems, the number of elements returned equals the number of RMS records in the file.

On Digital UNIX systems, the current size of the external file is the index of the last element in the file. The SIZE function returns the number of bytes in the file, divided by the open file's maximum element size.

14   **function** END_OF_FILE(FILE : **in** FILE_TYPE) **return** BOOLEAN;

15   Operates on a file of mode IN_FILE or INOUT_FILE. Returns TRUE
          if the current index exceeds the size of the external file; otherwise,
          returns FALSE.

16   The exception MODE_ERROR is raised if the mode of the given file is
          OUT_FILE.

17   **References** : count type 14.2, current index 14.2, current size 14.2, data_error
      exception 14.4, element 14.1, element_type 14.1, end_error exception 14.4, external file
      14.1, file 14.1, file mode 14.1, file_type 14.1, in_file 14.1, index 14.2, inout_file 14.1,
      mode_error exception 14.4, open file 14.1, positive_count 14.3, status_error exception
      14.4, use_error exception 14.4

      update a value 6.2, RMS record 14.1a

14.2.5 Specification of the Package Direct_IO

1   **with** IO_EXCEPTIONS;
     generic
        type ELEMENT_TYPE **is private**;
     package DIRECT_IO **is**

        type FILE_TYPE **is limited private**;

        type   FILE_MODE **is** (IN_FILE, INOUT_FILE, OUT_FILE);
        type   COUNT   **is range** 0 .. *implementation_defined*;
        subtype POSITIVE_COUNT **is** COUNT **range** 1 .. COUNT'LAST;

        -- File management

        procedure CREATE(FILE : **in out** FILE_TYPE;
                       MODE : **in** FILE_MODE := INOUT_FILE;
                       NAME : **in** STRING := "";
                       FORM : **in** STRING := "");

        -- NOTE: If ELEMENT_TYPE is an unconstrained type,
        -- a maximum element size must be specified
        -- in the FORM parameter of the CREATE procedure.

        procedure OPEN (FILE : **in out** FILE_TYPE;
                       MODE : **in** FILE_MODE;
                       NAME : **in** STRING;
                       FORM : **in** STRING := "");

        procedure CLOSE (FILE : **in out** FILE_TYPE);
        procedure DELETE(FILE : **in out** FILE_TYPE);
        procedure RESET (FILE : **in out** FILE_TYPE;
                       MODE : **in** FILE_MODE);
        procedure RESET (FILE : **in out** FILE_TYPE);

        function MODE   (FILE : **in** FILE_TYPE) **return** FILE_MODE;
        function NAME   (FILE : **in** FILE_TYPE) **return** STRING;
        function FORM   (FILE : **in** FILE_TYPE) **return** STRING;

```ada
      function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

      -- Input and output operations

      procedure READ (FILE : in FILE_TYPE;
                      ITEM : out ELEMENT_TYPE;
                      FROM : POSITIVE_COUNT);
      procedure READ (FILE : in FILE_TYPE;
                      ITEM : out ELEMENT_TYPE);

      procedure WRITE(FILE : in FILE_TYPE;
                      ITEM : in  ELEMENT_TYPE;
                      TO   : POSITIVE_COUNT);
      procedure WRITE(FILE : in FILE_TYPE;
                      ITEM : in ELEMENT_TYPE);

      procedure SET_INDEX(FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);

      function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
      function SIZE (FILE : in FILE_TYPE) return COUNT;

      function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

      -- Exceptions

      STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
      MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
      NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
      USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
      DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
      END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
      DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

   private
      -- implementation-dependent
   end DIRECT_IO;
```

2   **References** close procedure 14.2.1, count type 14.2, create procedure 14.2.1, data_
    error exception 14.4, default_mode 14.2.5, delete procedure 14.2.1, device_error
    exception 14.4, element_type 14.2.4, end_error exception 14.4, end_of_file function
    14.2.4, file_mode 14.2.5, file_type 14.2.4, form function 14.2.1, in_file 14.2.4, index
    function 14.2.4 14.2.1, io_exceptions package 14.4, is_open function
    14.2.1, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_
    error exception 14.4, open procedure 14.2.1, out_file 14.2.1, read procedure 14.2.4,
    set_index procedure 14.2.4, size function 14.2.4, status_error exception 14.4, use_error
    exception 14.4, write procedure 14.2.4 14.2.1

14.2a Relative and Indexed Files (OpenVMS Systems Only)

On OpenVMS systems, DEC Ada defines *relative access* and *indexed access* in addition to sequential and direct access. The corresponding file types and the associated operations are provided by the predefined generic packages RELATIVE_IO and INDEXED_IO. A file object to be used for relative access is called a *relative file*. A file object to be used for indexed access is called an *indexed file*.

Relative and indexed access are also provided in the predefined (nongeneric) packages RELATIVE_MIXED_IO and INDEXED_MIXED_IO, which allow values of different types to be mixed in a file. The operations provided by these packages are described in section 14.2b.

For relative access, the file is viewed as a set of fixed-length cells occupying consecutive positions in linear order. Cells can either be empty or can contain fixed- or variable-length elements. A cell is said to contain an element if an element has been written to that position (and not deleted), and a cell is said to be empty if an element has not been written to that position (or the last written element has been deleted).

The position of a cell is specified by its *index*, which is a number of the type COUNT (that is, it is in the range 0 .. INTEGER'LAST). The first cell in a relative file has an index of one. Like an open direct file, an open relative file has a *current index*, which is the index that will be used by the next read or write operation. When a relative file is opened, the current index is set to one. The current index of a relative file is a property of a file object, not of an external file. Also, the concept of size does not apply to relative files: end-of-file is true if all remaining elements, starting at the current index, are empty.

For indexed access, the file is viewed as a set of elements that are ordered by predefined *keys*. Each key has a number (a nonnegative integer) and a value. When the file is created, the keys of an element are defined in the form string to correspond to fields of that element, and cannot be changed thereafter. Each element has at least one primary key (numbered 0) and can have as many as 254 alternate keys (numbered 1 to 254).

The elements of the file can be accessed by any key. When a file is read, the element fields are searched until the key characteristics are matched. The read operation allows both *exact* and *inexact matching* of the key value. The type of matching can be specified by one of the values of the enumeration type RELATION_TYPE, which is defined as follows:

```
type RELATION_TYPE is (EQUAL_NEXT, EQUAL, NEXT);
```

The value EQUAL_NEXT corresponds to the case where a match will be greater than or equal to the key value if ascending keys are involved. The match will be less than or equal to the key value if descending keys are involved. The value EQUAL corresponds to the case where a match will be equal to the key value. The value NEXT corresponds to the case where a match will be greater than the key value if ascending keys are involved. The match will be less than the key value if descending keys are involved. If there are duplicate keys in the file, they are retrieved in the order in which they are written to the file.

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

An open relative or indexed file has a *current element*, which is the target element for the UPDATE and DELETE_ELEMENT operations. The current element is said to be defined after a successful READ, READ_EXISTING, READ_BY_KEY, or END_OF_FILE operation (if END_OF_FILE results in the reading of the next element of the file). The current element is said to be undefined after a successful CREATE, OPEN, CLOSE, DELETE, RESET, WRITE, UPDATE, UNLOCK, or DELETE_ELEMENT operation. The current element is said to be undefined after an unsuccessful READ, READ_EXISTING, READ_BY_KEY, WRITE, UPDATE, UNLOCK, DELETE_ ELEMENT, RESET (except when the exception MODE_ERROR is raised), or END_OF_FILE operation (if END_OF_FILE results in an attempt to read past the end of the external file).

All three file modes are allowed for relative and indexed files.

Depending on the sharing and access attributes specified in their form strings, relative and indexed files may be write shared. Element (RMS record) locking is automatically provided for the packages RELATIVE_IO and INDEXED_IO. When the current element in a relative or indexed file is defined (a successful read operation is performed), the element is locked until a subsequent read, write, update, delete, or unlock operation is performed; until the file is closed; or until an RMS operation on that file fails.

The purpose of locking is to prevent two (or more) tasks that share the same external file (by means of two different internal files) from interfering with each other's read and write operations. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information on sharing, file attributes, and record locking.

References: close procedure 14.2.1, count type 14.2, create procedure 14.2a.1 14.2b.1, delete procedure 14.2.1, delete_element procedure 14.2a.2 14.2a.4 14.2b.7 14.2b.9, element 14.1a, end_of_file function 14.2a.2 14.2a.4 14.2b.7 14.2b.9, enumeration type 3.5.1, external file 14.1, file object 14.1, form string 14.1 14.1b, generic package 12.1, indexed_io package 14.2a 14.2a.4, indexed_mixed_io package 14.2a 14.2b 14.2b.9, integer type 3.5.4, mode_error exception 14.4, open file 14.1, operation 3.3.3, package 7, range 3.5, read procedure 14.2a.2 14.2a.4 14.2b.7 14.2b.9, read_by_key procedure 14.2a.4 14.2b.9, read_existing procedure 14.2a.2 14.2b.7, relative_io package 14.2a 14.2a.2, relative_mixed_io package 14.2a 14.2b 14.2b.7, reset procedure 14.2.1 14.2a.1, task 9, unlock procedure 14.2a.2 14.2a.4 14.2b.7 14.2b.9, update procedure 14.2a.2 14.2a.4 14.2b.7 14.2b.9, RMS record 14.1a, write procedure 14.2a.2 14.2a.4 14.2b.7 14.2b.9

14.2a.1 File Management (OpenVMS Systems Only)

Except for the following differences, the procedures and functions for controlling relative and indexed access to external files containing values of the same type are the same as those for controlling sequential and direct access (see 14.2.1). The default mode for the CREATE procedures of both packages is INOUT_FILE.

For the package RELATIVE_IO, a maximum external element (record) size must be specified in the FORM parameter of the CREATE procedure if the package is instantiated with an unconstrained element type.

```
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE := INOUT_FILE;
                 NAME : in STRING := "";
                 FORM : in STRING := "");
--
-- Example:
--
-- procedure CREATE(FILE => MY_FILE;
--              FORM => "RECORD;"    &
--                      "SIZE  128" );
```

For the package INDEXED_IO, the FORM parameter of the CREATE procedure must be specified. There is no default. In particular, the FORM parameter must specify all information about the keys in the file to be created. If the package is instantiated with an unconstrained element type, a maximum external element (record) size must be specified.

```
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE := INOUT_FILE;
                 NAME : in STRING := "";
                 FORM : in STRING);
--
-- Example:
--
-- procedure CREATE(FILE => MY_FILE;
--                           FORM => "KEY 0;"            &
--                                   "LENGTH   5;"       &
--                                   "POSITION 0;"       &
--                                   "TYPE      STRING;");
--
```

Also for the package INDEXED_IO, the RESET procedures have a parameter for specifying the key number at which the file is to be reset. The default value for KEY_NUMBER is 0, which designates the primary key.

```
procedure RESET (FILE       : in FILE_TYPE;
                 MODE       : in FILE_MODE;
                 KEY_NUMBER : in INTEGER := 0);
procedure RESET (FILE       : in FILE_TYPE;
                 KEY_NUMBER : in INTEGER := 0);
```

References: create procedure 14.2.1, direct access 14.2, external file 14.1, file mode 14.1, form string 14.1 14.1b, function 6.5, indexed access 14.2a, inout_file 14.1, instantiation 12.3, key 14.2a, procedure 6.1, relative access 14.2a, relative_io package 14.2a 14.2a.2, reset procedure 14.2.1, sequential access 14.2, unconstrained type 3.3 3.3.2

14.2a.2 Relative Input-Output (OpenVMS Systems Only)

The operations available for relative input and output are described in this section. The exception STATUS_ERROR is raised if any of these operations is attempted for a file that is not open.

See section 14.2a.1 for information on the file management operations that are available for relative input and output.

```
procedure READ(FILE : in  FILE_TYPE;
               ITEM : out ELEMENT_TYPE;
               FROM : in  POSITIVE_COUNT);
procedure READ(FILE : in  FILE_TYPE;
               ITEM : out ELEMENT_TYPE);
```

Operates on a file of mode IN_FILE or INOUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter FROM. Then (for both forms) returns, in the parameter ITEM, the value of the element whose position is specified

by the current index of the given file. The element read becomes the current element, and the current index is increased by one.

The exception MODE_ERROR is raised if the current mode is OUT_FILE. The exception LOCK_ERROR is raised if the element to be read is locked. This error is possible only if the external file is being shared. The exception EXISTENCE_ERROR is raised if the element does not exist (the given cell is empty or the current index is beyond the end of the file).

```
procedure READ_EXISTING(FILE : in  FILE_TYPE;
                        ITEM : out ELEMENT_TYPE;
                        FROM : in  POSITIVE_COUNT);

procedure READ_EXISTING(FILE : in  FILE_TYPE;
                        ITEM : out ELEMENT_TYPE);
```

Operates on a file of mode IN_FILE or INOUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter FROM. Then (for both forms) starts at the current index and scans forward, skipping empty cells, and sets the current index to the first nonempty cell. Then returns, in the parameter ITEM, the value of the element whose position is specified by the current index of the given file. The element read becomes the current element, and the current index is increased by one.

The exception MODE_ERROR is raised if the mode of the given file is OUT_FILE. The exception LOCK_ERROR is raised if the element found is locked; this error is possible only if the external file is being shared. The exception EXISTENCE_ERROR is raised if the current index is beyond the end of the file, or if the end of the file is reached before an existing element (a nonempty cell) is found.

```
procedure WRITE(FILE : in FILE_TYPE;
                ITEM : in ELEMENT_TYPE;
                TO   : in POSITIVE_COUNT);

procedure WRITE(FILE : in FILE_TYPE;
                ITEM : in ELEMENT_TYPE);
```

Operates on a file of mode INOUT_FILE or OUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter TO. Then (for both forms) writes the value of the parameter ITEM to the cell whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

The exception MODE_ERROR is raised if the mode of the given file is IN_FILE. The exception USE_ERROR is raised if the element position in the file has already been written.

```
procedure UPDATE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);
```

Operates on a file of mode INOUT_FILE. Updates the current element of the given file with the value of the parameter ITEM.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation.

```
procedure UNLOCK(FILE : in FILE_TYPE);
```

Operates on a file of any mode. After this operation, the current element is undefined.

```
procedure DELETE_ELEMENT(FILE : in FILE_TYPE);
```

Operates on a file of mode INOUT_FILE. Deletes the current element in the file.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation.

```
procedure SET_INDEX(FILE : in FILE_TYPE;  TO : in POSITIVE_COUNT);
```

Operates on a file of any mode. Sets the current index of the given file to the index value given by the parameter TO.

```
function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
```

Operates on a file of any mode. Returns the current index of the given file.

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

Operates on a file of mode IN_FILE or INOUT_FILE. Returns TRUE if no cell, starting at the current index, contains an element; otherwise, returns FALSE.

The exception MODE_ERROR is raised if the current mode is OUT_FILE.

References: cell 14.2a, current element 14.2a, current index 14.2a, element 14.1a, end of file 14.2a, existence_error exception 14.4, file 14.1, file mode 14.1, file sharing 14.2a, index 14.2a, in_file 14.1, inout_file 14.1, lock_error exception 14.4, locking 14.2a, mode_error exception 14.4, open file 14.1, out_file 14.1, relative access 14.2a, relative

file 14.2a, status_error exception 14.4, undefined current element 14.2a, use_error exception 14.4

14.2a.3 Specification of the Package Relative_IO (OpenVMS Systems Only)

```
with IO_EXCEPTIONS;
with AUX_IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package RELATIVE_IO is

   type    FILE_TYPE       is limited private;
   type    FILE_MODE       is (IN_FILE, INOUT_FILE, OUT_FILE);
   type    COUNT           is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT  is COUNT range 1 .. COUNT'LAST;

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := INOUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "");

   -- NOTE: If ELEMENT_TYPE is an unconstrained type,
   -- a maximum element (or record) size must be specified
   -- in the FORM parameter of the CREATE procedure.

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE;
                    NAME : in STRING;
                    FORM : in STRING := "");

   procedure CLOSE (FILE :  in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);

   function MODE    (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME    (FILE : in FILE_TYPE) return STRING;
   function FORM    (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

   -- Input and output operations

   procedure READ  (FILE : in  FILE_TYPE;
                    ITEM : out ELEMENT_TYPE;
                    FROM : in  POSITIVE_COUNT);
   procedure READ  (FILE : in  FILE_TYPE;
                    ITEM : out ELEMENT_TYPE);
```

```
    procedure READ_EXISTING(FILE : in  FILE_TYPE;
                            ITEM : out ELEMENT_TYPE;
                            FROM : in  POSITIVE_COUNT);
    procedure READ_EXISTING(FILE : in  FILE_TYPE;
                            ITEM : out ELEMENT_TYPE);

    procedure WRITE (FILE : in FILE_TYPE;
                     ITEM : in ELEMENT_TYPE;
                     TO   : in POSITIVE_COUNT);
    procedure WRITE (FILE : in FILE_TYPE;
                     ITEM : in ELEMENT_TYPE);

    procedure UPDATE(FILE : in FILE_TYPE;
                     ITEM : in ELEMENT_TYPE);

    procedure UNLOCK(FILE : in FILE_TYPE);

    procedure DELETE_ELEMENT(FILE : in FILE_TYPE);

    procedure SET_INDEX(FILE : in FILE_TYPE;
                        TO   : in POSITIVE_COUNT);

    function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;

    function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

    -- Exceptions

    STATUS_ERROR    : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR      : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR      : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR       : exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR    : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR       : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR      : exception renames IO_EXCEPTIONS.DATA_ERROR;
    LOCK_ERROR      : exception renames AUX_IO_EXCEPTIONS.LOCK_ERROR;
    EXISTENCE_ERROR : exception
       renames AUX_IO_EXCEPTIONS.EXISTENCE_ERROR;

private
    -- implementation-dependent
end RELATIVE_IO;
```

References: aux_io_exceptions package 14.4, close procedure 14.2.1, count type 14.2, create procedure 14.2a.1, data_error exception 14.4, delete procedure 14.2.1, delete_element procedure 14.2a.2, device_error exception 14.4, element_type 14.1, end_error exception 14.4, end_of_file function 14.2a.2, existence_error exception 14.4, file_mode 14.1, file_type 14.1, form function 14.2.1, index function 14.2a.2, integer type 3.5.4, io_exceptions package 14.4, is_open function 14.2.1, lock_error exception 14.4, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_error exception 14.4, open procedure 14.2.1, read procedure 14.2a.2, read_existing procedure 14.2a.2, reset procedure 14.2.1, set_index procedure 14.2a.2, status_error exception 14.4, unconstrained type 3.3 3.3.2, unlock procedure 14.2a.2, update procedure 14.2a.2, use_error exception 14.4, write procedure 14.2a.2

14.2a.4 Indexed Input-Output (OpenVMS Systems Only)

The operations available for indexed input and output are described in this
section. The exception STATUS_ERROR is raised if any of these operations is
attempted for a file that is not open.

See section 14.2a.1 for information on the file management operations that are
available for indexed input and output.

The package INDEXED_IO provides a generic READ_BY_KEY procedure,
which defines input for the given element type. This procedure must be
instantiated with an actual type parameter for the generic parameter
KEY_TYPE. It can be instantiated with or without a value for the generic
parameter DEFAULT_KEY_NUMBER. The range of values for DEFAULT_
KEY_NUMBER is 0 to 254. A value of 0 (the default) designates the primary
key.

If the RELATION parameter to the READ_BY_KEY procedure is specified,
it must have a value of EQUAL_NEXT, EQUAL, or NEXT (the default is
EQUAL).

```
procedure READ (FILE : in  FILE_TYPE; ITEM : out  ELEMENT_TYPE);

generic
    type KEY_TYPE is private;
    DEFAULT_KEY_NUMBER : INTEGER := 0;
procedure READ_BY_KEY(FILE        : in  FILE_TYPE;
                      ITEM        : out ELEMENT_TYPE;
                      KEY         : in  KEY_TYPE;
                      KEY_NUMBER : in   INTEGER :=
                                        DEFAULT_KEY_NUMBER;
                      RELATION    : in  RELATION_TYPE := EQUAL);
```

Operates on a file of mode IN_FILE or INOUT_FILE. In the case of
the first form, returns, in the parameter ITEM, the value of the next
element, according to the most recent key and relation information. In
the case of the second form, returns, in the parameter ITEM, the value
of the element specified by the given key information; KEY gives the
key value; KEY_NUMBER designates a primary (0) or alternate key (1
to 254); and RELATION determines the kind of match to be made for
the key value.

For both forms, the element read becomes the current element. In the
case of the first form, the next sequential element becomes the next
element, according to the most recent key and relation information. In
the case of the second form, the next sequential element that matches
the key and relation information specified becomes the next element. If
neither the key nor the relation information changes from one READ_

BY_KEY operation to the next, the same element will continue to be read.

The exception MODE_ERROR is raised if the current mode is OUT_FILE.

The exception END_ERROR is raised if an attempt is made to read past the end of the file by the first form.

The exception LOCK_ERROR is raised if the element to be read is locked. This error is possible only if the external file is being shared. The exception EXISTENCE_ERROR is raised if the element does not exist. The exception KEY_ERROR is raised if the size of the given key is not a multiple of eight bits.

procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

Operates on a file of mode INOUT_FILE or OUT_FILE. Gives the value of the parameter ITEM to the element whose position in the given file is specified by the key information contained within the value of ITEM.

The exception MODE_ERROR is raised if the current mode is IN_FILE.

The exception USE_ERROR is raised if the element position in the file has already been written.

The exception KEY_ERROR is raised if a key has been duplicated and if duplicates are not allowed by the external file.

procedure UPDATE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

Operates on a file of mode INOUT_FILE. Updates the current element of the given file with the value of the parameter ITEM.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE.

The exception USE_ERROR is raised if the current element is undefined at the start of this operation or if some key specification in ITEM violates the external file attributes defined for that key.

The exception KEY_ERROR is raised if a key has been changed or duplicated and if changes or duplicates are not allowed by the external file.

procedure UNLOCK(FILE : in FILE_TYPE);

Operates on a file of any mode. After this operation, the current element is undefined.

```
procedure DELETE_ELEMENT(FILE : in FILE_TYPE);
```

> Operates on a file of mode INOUT_FILE. Deletes the current element of the file.

> The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation.

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

> Operates on a file of mode IN_FILE or INOUT_FILE. Returns TRUE if there are no more elements (according to the most recent key and relation information) starting at the next element in the file; otherwise, returns FALSE.

> The exception MODE_ERROR is raised if the current mode is OUT_FILE.

References: current element 14.2a, element 14.1a, element_type 14.1, end_error exception 14.4, existence_error exception 14.4, external file 14.1, file 14.1, file mode 14.1, file sharing 14.2a, generic actual parameter 12.3, generic formal parameter 12.3, generic procedure 12.1, in_file 14.1, inout_file 14.1, instantiation 12.3, key 14.2a, key_error exception 14.4, lock_error exception 14.4, locking 14.2a, mode_error exception 14.4, next element 14.2a, out_file 14.1, range 3.5, relation_type 14.2a, read_by_key procedure 14.2a.4, status_error exception 14.4, undefined current element 14.2a, use_error exception 14.4

14.2a.5 Specification of the Package Indexed_IO (OpenVMS Systems Only)

```
with IO_EXCEPTIONS;
with AUX_IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package INDEXED_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

   type RELATION_TYPE is (EQUAL_NEXT, EQUAL, NEXT);

   function GREATER return RELATION_TYPE
      renames NEXT;
   function GREATER_EQUAL return RELATION_TYPE
      renames EQUAL_NEXT;

   -- File management
```

```ada
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE := INOUT_FILE;
                 NAME : in STRING := "";
                 FORM : in STRING);
-- NOTE: All information about the keys must be provided
-- in the FORM parameter of the CREATE procedure.
-- If ELEMENT_TYPE is an unconstrained type, a
-- maximum element (or record) size must also be specified.
procedure OPEN  (FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE;
                 NAME : in STRING;
                 FORM : in STRING := "");
procedure CLOSE (FILE :  in out FILE_TYPE);
procedure DELETE(FILE : in out FILE_TYPE);
procedure RESET (FILE       : in FILE_TYPE;
                 MODE       : in FILE_MODE;
                 KEY_NUMBER : in INTEGER := 0);
procedure RESET (FILE       : in FILE_TYPE;
                 KEY_NUMBER : in INTEGER := 0);

function MODE   (FILE : in FILE_TYPE) return FILE_MODE;
function NAME   (FILE : in FILE_TYPE) return STRING;
function FORM   (FILE : in FILE_TYPE) return STRING;

function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- Input and output operations

procedure READ  (FILE : in  FILE_TYPE;
                 ITEM : out  ELEMENT_TYPE);

generic
    type KEY_TYPE is private;
    DEFAULT_KEY_NUMBER : INTEGER := 0;
procedure READ_BY_KEY (FILE       : in  FILE_TYPE;
                       ITEM       : out ELEMENT_TYPE;
                       KEY        : in  KEY_TYPE;
                       KEY_NUMBER : in  INTEGER :=
                                           DEFAULT_KEY_NUMBER;
                       RELATION   : in  RELATION_TYPE := EQUAL);

procedure WRITE (FILE : in FILE_TYPE;
                 ITEM : in ELEMENT_TYPE);

procedure UPDATE(FILE : in FILE_TYPE;
                 ITEM : in ELEMENT_TYPE);

procedure UNLOCK(FILE : in FILE_TYPE);

procedure DELETE_ELEMENT(FILE : in FILE_TYPE);

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

```
    -- Exceptions

    STATUS_ERROR      : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR        : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR        : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR         : exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR      : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR         : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR        : exception renames IO_EXCEPTIONS.DATA_ERROR;
    LOCK_ERROR        : exception renames AUX_IO_EXCEPTIONS.LOCK_ERROR;
    KEY_ERROR         : exception renames AUX_IO_EXCEPTIONS.KEY_ERROR;
    EXISTENCE_ERROR : exception
        renames AUX_IO_EXCEPTIONS.EXISTENCE_ERROR;

private
    -- implementation-dependent
end INDEX_IO;
```

References: aux_io_exceptions package 14.4, close procedure 14.2.1, create procedure 14.2a.1, data_error exception 14.4, delete procedure 14.2.1, delete_element procedure 14.2a.4, device_error exception 14.4, element_type 14.1, end_error exception 14.4, end_of_file function 14.2a.4, existence_error exception 14.4, file_mode 14.1, file_type 14.1, form function 14.2.1, form parameter 14.1, io_exceptions package 14.4, is_open function 14.2.1, key 14.2a, key_error exception 14.4, lock_error exception 14.4, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_error exception 14.4, open procedure 14.2.1, read procedure 14.2a.4, read_by_key procedure 14.2a.4, record size 14.2b, relation_type 14.2a, reset procedure 14.2a.1, status_error exception 14.4, unconstrained type 3.3 3.3.2, unlock procedure 14.2a.4, update procedure 14.2a.4, use_error exception 14.4, write procedure 14.2a.4

14.2b Mixed-Type Input-Output

The following DEC Ada predefined packages provide types and operations for files consisting of elements that can contain zero or more items of various types:

SEQUENTIAL_MIXED_IO
DIRECT_MIXED_IO
RELATIVE_MIXED_IO
INDEXED_MIXED_IO

Section 14.7a gives an example of using a mixed-type input-output package. The packages RELATIVE_MIXED_IO and INDEXED_MIXED_IO are available on OpenVMS systems only.

Each file opened by one of these mixed-type input-output packages has an associated file buffer maintained by the package. The size of the file buffer is limited by the maximum element size of the external file. The maximum

element size is controlled by the FORM parameter in a CREATE or OPEN procedure as follows:

- On OpenVMS systems, the maximum element size is controlled by a FORM parameter that includes the string `RECORD; SIZE max_record_size`.

- On Digital UNIX systems, the maximum element size is controlled by a FORM parameter that includes the string `ELEMENT_SIZE => max_element_size`.

(See 14.1b.1 for more information on specifying FORM parameters.)

Element input-output operations, such as READ and WRITE, provide the means for reading a file element into the file buffer (and reading an element of the associated external file) or writing the contents of the file buffer into an element (and writing an element of the associated external file).

Item input-output operations, such as GET and PUT, provide the means for transferring an item (value) from or to the file buffer. Every item transfer begins on a byte boundary and involves the number of bits required by the type of the item. The *position* or *current position* in the file buffer is thus a byte offset from the beginning of the buffer. All of the predefined mixed-type input-output packages provide the same item input-output operations. See section 14.2b.2.

On OpenVMS systems, element (RMS record) locking is provided for the packages RELATIVE_MIXED_IO and INDEXED_MIXED_IO. A brief explanation of how locking affects relative and indexed files containing elements of the same type is given at the end of section 14.2a. The same information applies to relative and indexed files containing elements of mixed types.

References: direct file 14.2, element 14.1a, locking 14.2a, external file 14.1, indexed file 14.2, open file 14.1, package 7, relative file 14.2, sequential file 14.2, RMS record 14.1a

14.2b.1 File Management

Except for the following differences, the procedures and functions for controlling access to files containing mixed-type values are the same as those for controlling access to files containing values of the same type (see 14.2.1 and 14.2a.1). The default mode for the CREATE procedure of the package SEQUENTIAL_MIXED_IO is OUT_FILE. INOUT_FILE is the default mode for the CREATE procedures of the packages DIRECT_MIXED_IO, RELATIVE_MIXED_IO, and INDEXED_MIXED_IO.

For the packages DIRECT_MIXED_IO and RELATIVE_MIXED_IO, a maximum element size must be specified in the FORM parameter of the CREATE procedures.

```
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE := INOUT_FILE;
                 NAME : in STRING := "";
                 FORM : in STRING := "");
--
-- OpenVMS example:
--
-- procedure CREATE(FILE => MY_FILE;
--                  FORM => "RECORD;"    &
--                          "SIZE  128" );
--
-- Digital UNIX example:
--
-- procedure CREATE(FILE => MY_FILE;
--                  FORM => "ELEMENT_SIZE => 128");
```

For the package INDEXED_MIXED_IO, all information about the file keys must be specified in the FORM parameter of the CREATE procedure.

```
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE := INOUT_FILE;
                 NAME : in STRING := "";
                 FORM : in STRING);
--
-- Example:
--
-- procedure CREATE(FILE => MY_FILE;
--                  FORM => "KEY 0;"            &
--                          "LENGTH   5;"    &
--                          "POSITION 0;"    &
--                          "TYPE     STRING;");
--
```

References: direct_mixed_io package 14.2 14.2b 14.2b.5, file 14.1, file mode 14.1, form parameter 14.1 14.1b, function 6.5, indexed_mixed_io package 14.2a 14.2b 14.2b.9, inout_file 14.1, key 14.2a, out_file 14.1, procedure 6.1, relative_mixed_io package 14.2a 14.2b 14.2b.7, sequential_mixed_io package 14.2 14.2b 14.2b.3

14.2b.2 Item Input-Output

The same item input-output operations are available in all of the DEC Ada mixed-type input-output packages. This section describes the operations that provide item input and output to and from the file buffer.

All of the mixed-type packages provide generic GET_ITEM, GET_ARRAY, and PUT_ITEM procedures, which define buffer input and output for given item and item-array types.

The GET_ITEM and PUT_ITEM procedures must be instantiated with an actual type parameter for the generic parameter ITEM_TYPE. The GET_ ARRAY procedure must be instantiated with actual type parameters for the generic parameters ITEM_TYPE (to determine the array component types), INDEX (to determine how the array is indexed), and ITEM_ARRAY (to determine the array to be output).

```
generic
    type ITEM_TYPE is private;
procedure GET_ITEM(FILE : in FILE_TYPE; ITEM : out ITEM_TYPE);
```

> Operates on a file buffer for a file of mode IN_FILE or INOUT_FILE. Gets an item from the file buffer at the current position, and returns its value in the parameter ITEM. The current position in the file buffer is updated to the position of the next item.
>
> The exception MODE_ERROR is raised if the current mode of the given file is OUT_FILE. The exception LAYOUT_ERROR is raised if no more items can be read from the file buffer.

```
generic
    type ITEM_TYPE is private;
    type INDEX is (<>);
    type ITEM_ARRAY is array (INDEX range <>) of ITEM_TYPE;
procedure GET_ARRAY(FILE  : in FILE_TYPE;
                    ITEMS : out ITEM_ARRAY;
                    LAST  : out INDEX);
```

> Operates on a file buffer for a file of mode IN_FILE or INOUT_FILE. Gets the remaining items in the file buffer and returns their values in the parameter ITEMS. Sets LAST to the index of the last element of ITEMS that is updated.
>
> Reading stops when the array ITEMS is full or when no more items can be read from the file buffer. The current position in the file buffer is updated to the position after the last position read.
>
> If no items are read, returns in LAST an index value that is one less than ITEMS'FIRST.
>
> The exception MODE_ERROR is raised if the current mode of the given file is OUT_FILE.

```
generic
    type ITEM_TYPE is private;
procedure PUT_ITEM(FILE : in FILE_TYPE; ITEM : in ITEM_TYPE);
```

> Operates on a file buffer for a file of mode OUT_FILE or INOUT_FILE. The value of the parameter ITEM is written into the file buffer at the

current position, and the current position is updated to the position of the next item in the file buffer.

The exception MODE_ERROR is raised if the current mode of the given file is IN_FILE. The exception LAYOUT_ERROR is raised if the current position exceeds the file buffer size. The file buffer size is limited by the maximum size of an element in the external file.

function END_OF_BUFFER(FILE : **in** FILE_TYPE) **return** BOOLEAN;

Operates on a file buffer for a file of mode IN_FILE or INOUT_FILE. Returns TRUE if there are no more items to be read from the file buffer. Returns FALSE otherwise.

The exception MODE_ERROR is raised if the current mode of the given file is OUT_FILE.

procedure SET_POSITION(FILE : **in** FILE_TYPE;
                    TO : **in** POSITIVE_COUNT);

Operates on a file buffer for a file of any mode. Sets the current position of the file buffer to the position specified by the value of TO.

function POSITION(FILE : **in** FILE_TYPE) **return** POSITIVE_COUNT;

Operates on a file buffer for a file of any mode. Returns the current position in the file buffer.

function MAX_ELEMENT_SIZE(FILE : **in** FILE_TYPE) **return** COUNT;

Operates on a file of any mode. Returns the maximum element size (in bytes) specified for the external file. The maximum element size, in turn, defines the limits for the size of the file buffer. A value of zero indicates that there is no maximum element size associated with the file.

On OpenVMS systems, a maximum element size can be specified for a file (when it is created) with a FORM parameter value that includes the string "RECORD; SIZE max_record_size". On OpenVMS systems, the buffer size is limited only by the maximum size of an RMS record.

On Digital UNIX systems, a maximum element size can be specified for a file with a FORM parameter value that includes the string "ELEMENT_SIZE => max_element_size".

function ELEMENT_SIZE(FILE : **in** FILE_TYPE) **return** COUNT;

Operates on a file buffer for a file of mode IN_FILE or INOUT_FILE. Returns the size (in bytes) of the element (record) last read into the file

buffer. If no elements have ever been read into the file buffer, a value of zero is returned.

The ELEMENT_SIZE function can be used in combination with the POSITION function to determine the number of bytes remaining in the file buffer (of the last element read): ELEMENT_SIZE (F) – POSITION (F) + 1, for any file F.

The exception MODE_ERROR is raised if the mode is not IN_FILE or INOUT_FILE.

References: buffer size 14.2b, current position 14.2b, direct_mixed_io package 14.2 14.2b 14.2b.3, external file 14.1, file 14.1, file buffer 14.2b, file mode 14.1, indexed_mixed_io package 14.2a 14.2b 14.2b.9, in_file 14.1, inout_file 14.1, item 14.2b, layout_error exception 14.4, mode_error exception 14.4, out_file 14.1, relative_mixed_io package 14.2a 14.2b 14.2b.7, sequential_mixed_io package 14.2 14.2b 14.2b.3, RMS record 14.1a

14.2b.3 Sequential Mixed Input-Output

The operations available for sequential mixed-type input and output are described in this section. The exception STATUS_ERROR is raised if any of these operations is attempted for a file that is not open.

See sections 14.2b.1 and 14.2b.2 for information on the file management and item input-output operations that are available for sequential mixed input and output.

```
procedure READ(FILE : in FILE_TYPE);
```

Operates on a file of mode IN_FILE. Reads the next sequential element from the given file into the file buffer.

The exception MODE_ERROR is raised if the mode is not IN_FILE. The exception END_ERROR is raised if an attempt is made to read past the end of the file.

```
procedure WRITE(FILE : in FILE_TYPE);
```

Operates on a file of mode OUT_FILE. Writes the file buffer to the given file as a new element. If the external file element format is fixed and the current position in the file buffer does not indicate the end of the buffer, the rest of the buffer is filled with zero bits before being written to the file.

The exception MODE_ERROR is raised if the mode is not OUT_FILE.

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

Operates on a file of mode IN_FILE. Returns TRUE if no more elements can be read from the given file; otherwise returns FALSE.

The exception MODE_ERROR is raised if the mode is not IN_FILE.

References: current position 14.2b, element 14.1a, end_error exception 14.4, external file 14.1, file 14.1, file buffer 14.2b, file mode 14.1, in_file 14.1, mixed-type file 14.1a, mode_error exception 14.4, open file 14.1, out_file 14.1, sequential file 14.2, status_error exception 14.4

14.2b.4 Specification of the Package Sequential_Mixed_IO

```
with IO_EXCEPTIONS;
package SEQUENTIAL_MIXED_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := OUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "");

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE;
                    NAME : in STRING;
                    FORM : in STRING := "");

   procedure CLOSE (FILE :  in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);

   function MODE    (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME    (FILE : in FILE_TYPE) return STRING;
   function FORM    (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

   -- Item input and output operations

   generic
      type ITEM_TYPE is private;
   procedure GET_ITEM (FILE : in FILE_TYPE; ITEM : out ITEM_TYPE);
```

```ada
generic
   type ITEM_TYPE is private;
   type INDEX is (<>);
   type ITEM_ARRAY is array (INDEX range <>) of ITEM_TYPE;

procedure GET_ARRAY(FILE  : in FILE_TYPE;
                    ITEMS : out ITEM_ARRAY;
                    LAST  : out INDEX);

generic
   type ITEM_TYPE is private;
procedure PUT_ITEM (FILE : in FILE_TYPE; ITEM : in ITEM_TYPE);
function END_OF_BUFFER(FILE : in FILE_TYPE) return BOOLEAN;

procedure SET_POSITION(FILE : in FILE_TYPE;
                         TO : in POSITIVE_COUNT);

function POSITION(FILE : in FILE_TYPE) return POSITIVE_COUNT;

function MAX_ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

function ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

-- Element input and output operations

procedure READ (FILE : in FILE_TYPE);
procedure WRITE(FILE : in FILE_TYPE);

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

-- Exceptions

STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

private
   -- implementation-dependent
end SEQUENTIAL_MIXED_IO;
```

References: close procedure 14.2.1, count type 14.2, create procedure 14.2.1, data_ error exception 14.4, delete procedure 14.2.1, device_error exception 14.4, element_ size function 14.2b.2, end_error exception 14.4, end_of_buffer function 14.2b.2, end_ of_file function 14.2b.3, file_mode 14.1, file_type 14.1, form function 14.2.1, get_ array procedure 14.2b.2, get_item procedure 14.2b.2, io_exceptions package 14.4, is_open function 14.2.1, item_array 14.2b.2, item_type 14.2b.2, max_element_size function 14.2b.2, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_error exception 14.4, open procedure 14.2.1, position function 14.2b.2, put_item procedure 14.2b.2, read procedure 14.2b.3, reset procedure 14.2.1, set_ position procedure 14.2b.2, status_error exception 14.4, use_error exception 14.4, write procedure 14.2b.3

14.2b.5 Direct Mixed Input-Output

The operations available for direct mixed-type input and output are described in this section. The exception STATUS_ERROR is raised if any of these operations is attempted for a file that is not open.

See sections 14.2b.1 and 14.2b.2 for information on the file management and element input-output operations that are available for direct mixed input and output.

```
procedure READ (FILE : in FILE_TYPE;
                FROM : in POSITIVE_COUNT);

procedure READ (FILE : in FILE_TYPE);
```

Operates on a file of mode IN_FILE or INOUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter FROM. Then (for both forms) returns, in the file buffer, the value of the element whose position is specified by the current index of the file; finally, increases the current index by one.

The exception MODE_ERROR is raised if the mode of the given file is OUT_FILE. The exception END_ERROR is raised if the index to be used exceeds the size of the external file.

```
procedure WRITE (FILE : in FILE_TYPE;
                 TO   : in POSITIVE_COUNT);

procedure WRITE (FILE : in FILE_TYPE);
```

Operates on a file of mode INOUT_FILE or OUT_FILE. In the case of the first form, sets the index of the given file to the index value given by the parameter TO. Then (for both forms) writes the file buffer to the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

If the current position in the file buffer does not indicate the end of the buffer, the rest of the file buffer is filled with zero bits before being written to the file.

The exception MODE_ERROR is raised if the mode of the given file is IN_FILE.

```
procedure SET_INDEX(FILE : in FILE_TYPE;
                    TO   : in POSITIVE_COUNT);
```

Operates on a file of any mode. Sets the current index of the given file to the index value given by the parameter TO. The value can exceed the size of the file.

```
function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
```

> Operates on a file of any mode. Returns the current index of the given file.

```
function SIZE(FILE : in FILE_TYPE) return COUNT;
```

> Operates on a file of any mode. Returns the number of elements in the file.

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

> Operates on a file of mode IN_FILE or INOUT_FILE. Returns TRUE if the current index exceeds the size of the external file; otherwise returns FALSE.

> The exception MODE_ERROR is raised if the current mode is OUT_FILE.

References: current index 14.2a, current position 14.2b, direct file 14.2, element 14.1a, end_error exception 14.4, external file 14.1, file 14.1, file buffer 14.2b, file mode 14.1, index 14.2, in_file 14.1, inout_file 14.1, mixed-type file 14.1a, mode_error exception 14.4, open file 14.1, out_file 14.1, status_error exception 14.4, RMS record 14.1a

14.2b.6 Specification of the Package Direct_Mixed_IO

```
with IO_EXCEPTIONS;
package DIRECT_MIXED_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
   type COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := INOUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "");

   -- NOTE: A maximum element size must be specified
   -- in the FORM parameter of the CREATE procedure.

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE;
                    NAME : in STRING;
                    FORM : in STRING := "");

   procedure CLOSE (FILE :  in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);
```

```ada
   procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);

   function MODE    (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME    (FILE : in FILE_TYPE) return STRING;
   function FORM    (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

   -- Item input and output operations
   generic
      type ITEM_TYPE is private;
   procedure GET_ITEM (FILE : in FILE_TYPE; ITEM : out ITEM_TYPE);

   generic
      type ITEM_TYPE is private;
      type INDEX is (<>);
      type ITEM_ARRAY is array (INDEX range <>) of ITEM_TYPE;
   procedure GET_ARRAY(FILE  : in FILE_TYPE;
                       ITEMS : out ITEM_ARRAY;
                       LAST  : out INDEX);
   generic
      type ITEM_TYPE is private;
   procedure PUT_ITEM (FILE : in FILE_TYPE; ITEM : in ITEM_TYPE);

   function END_OF_BUFFER(FILE : in FILE_TYPE) return BOOLEAN;

   procedure SET_POSITION(FILE : in FILE_TYPE;
                            TO : in POSITIVE_COUNT);

   function POSITION  (FILE : in FILE_TYPE) return POSITIVE_COUNT;

   function MAX_ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

   function ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

   -- Element input and output operations
   procedure READ (FILE : in FILE_TYPE;
                   FROM : in POSITIVE_COUNT);
   procedure READ (FILE : in FILE_TYPE);

   procedure WRITE(FILE : in FILE_TYPE;
                   TO   : in POSITIVE_COUNT);
   procedure WRITE(FILE : in FILE_TYPE);

   procedure SET_INDEX(FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);

   function INDEX (FILE : in FILE_TYPE) return POSITIVE_COUNT;

   function SIZE  (FILE : in FILE_TYPE) return COUNT;

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

   -- Exceptions
```

```
          STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
          MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
          NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
          USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
          DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
          END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
          DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

      private
          -- implementation-dependent
      end DIRECT_MIXED_IO;
```

References: close procedure 14.2.1, count type 14.2, create procedure 14.2b.1, data_
error exception 14.4, delete procedure 14.2.1, device_error exception 14.4, element_size
function 14.2b.2, end_error exception 14.4, end_of_buffer function 14.2b.2, end_of_
file function 14.2b.5, file_mode 14.1, file_type 14.1, form function 14.2.1, get_array
procedure 14.2b.2, get_item procedure 14.2b.2, index function 14.2b.5, io_exceptions
package 14.4, is_open function 14.2.1, item_array 14.2b.2, item_type 14.2b.2, max_
element_size function 14.2b.2, mode function 14.2.1, mode_error exception 14.4, name
function 14.2.1, name_error exception 14.4, open procedure 14.2.1, position function
14.2b.2, put_item procedure 14.2b.2, read procedure 14.2b.5, reset procedure 14.2.1,
set_index procedure 14.2b.5, set_position procedure 14.2b.2, size function 14.2b.5,
status_error exception 14.4, use_error exception 14.4, write procedure 14.2b.5

14.2b.7 Relative Mixed Input-Output (OpenVMS Systems Only)

The operations available for relative mixed-type input and output are described
in this section. The exception STATUS_ERROR is raised if any of these
operations is attempted for a file that is not open.

See sections 14.2b.1 and 14.2b.2 for information on the file management and
element input-output operations that are available for relative mixed input and
output.

```
procedure READ (FILE : in FILE_TYPE;
                FROM : in POSITIVE_COUNT);

procedure READ (FILE : in FILE_TYPE);
```

Operates on a file of mode IN_FILE or INOUT_FILE. In the case of
the first form, sets the current index of the given file to the index value
given by the parameter FROM. Then (for both forms) returns, in the
file buffer, the value of the element whose position is specified by the
current index of the given file. The element read becomes the current
element, and the current index is increased by one.

The exception MODE_ERROR is raised if the current mode is OUT_
FILE. The exception LOCK_ERROR is raised if the element to be read
is locked; this error is possible only if the external file is being shared.
The exception EXISTENCE_ERROR is raised if the element does not

exist (that is, the given cell is empty or the current index is beyond the end of the file).

```
procedure READ_EXISTING (FILE : in FILE_TYPE;
                         FROM : in POSITIVE_COUNT);

procedure READ_EXISTING (FILE : in FILE_TYPE);
```

Operates on a file of mode IN_FILE or INOUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter FROM. Then (for both forms) starts at the current index and scans forward, skipping empty cells, and sets the current index to the first nonempty cell. Then returns, in the file buffer, the value of the element whose position is specified by the current index of the given file. The element read becomes the current element, and the current index is increased by one.

The exception MODE_ERROR is raised if the mode of the given file is OUT_FILE. The exception LOCK_ERROR is raised if the element to be read is locked; this error is possible only if the external file is being shared. The exception EXISTENCE_ERROR is raised if the current index is beyond the end of the file, or if the end of the file is reached before an existing element (nonempty cell) is found.

```
procedure WRITE (FILE : in FILE_TYPE;
                 TO   : in POSITIVE_COUNT);

procedure WRITE (FILE : in FILE_TYPE);
```

Operates on a file of mode INOUT_FILE or OUT_FILE. In the case of the first form, sets the current index of the given file to the index value given by the parameter TO. Then (for both forms) writes the contents of the file buffer to the cell whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

If the format of the associated external file is fixed and the current position in the file buffer does not indicate the end of the buffer, the rest of the file buffer is filled with zero bits before being written to the file.

The exception MODE_ERROR is raised if the mode of the given file is IN_FILE. The exception USE_ERROR is raised if the element position in the file has already been written.

```
procedure UPDATE(FILE : in FILE_TYPE);
```

Operates on a file of mode INOUT_FILE. Updates the current element with the contents of the file buffer.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation.

procedure UNLOCK (FILE : in FILE_TYPE);

Operates on a file of any mode. After this operation, the current element is undefined.

procedure DELETE_ELEMENT(FILE : in FILE_TYPE);

Operates on a file of mode INOUT_FILE. Deletes the current element in the file.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation.

procedure SET_INDEX(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);

Operates on a file of any mode. Sets the current index of the given file to the index value specified by the parameter TO.

function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;

Operates on a file of any mode. Returns the current index of the given file.

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

Operates on a file of mode IN_FILE or INOUT_FILE. Returns TRUE if no cell, starting at the current index, contains an element; otherwise returns FALSE.

The exception MODE_ERROR is raised if the current mode is OUT_FILE.

References: cell 14.2a, current element 14.2a, current index 14.2a, current mode 14.1, current position 14.2b, element 14.1a, end of file 14.2a, existence_error exception 14.4, file 14.1, file buffer 14.2b, file mode 14.1, file sharing 14.2a, index 14.2a, in_file 14.1, inout_file 14.1, lock_error exception 14.4, locking 14.2a, mixed-type file 14.1a, mode_error exception 14.4, open file 14.1, out_file 14.1, relative file 14.2a, status_error exception 14.4, undefined current element 14.2a, use_error exception 14.4

14.2b.8 Specification of the Package Relative_Mixed_IO (OpenVMS Systems Only)

```ada
with IO_EXCEPTIONS;
with AUX_IO_EXCEPTIONS;
package RELATIVE_MIXED_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

   type COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := INOUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "");

   -- NOTE: A maximum element (or record) size must be
   -- specified in the FORM parameter of the CREATE
   -- procedure.

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE;
                    NAME : in STRING;
                    FORM : in STRING := "");

   procedure CLOSE (FILE : in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);

   function MODE    (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME    (FILE : in FILE_TYPE) return STRING;
   function FORM    (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

   -- Item input and output operations

   generic
      type ITEM_TYPE is private;
   procedure GET_ITEM(FILE : in FILE_TYPE; ITEM : out ITEM_TYPE);

   generic
      type ITEM_TYPE is private;
      type INDEX is (<>);
      type ITEM_ARRAY is array (INDEX range <>) of ITEM_TYPE;
   procedure GET_ARRAY(FILE  : in FILE_TYPE;
                       ITEMS : out ITEM_ARRAY;
                       LAST  : out INDEX);
```

```ada
   generic
      type ITEM_TYPE is private;
   procedure PUT_ITEM (FILE : in FILE_TYPE; ITEM : in ITEM_TYPE);

   function END_OF_BUFFER(FILE : in FILE_TYPE) return BOOLEAN;

   procedure SET_POSITION(FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);

   function POSITION  (FILE : in FILE_TYPE) return POSITIVE_COUNT;

   function MAX_ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

   function ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

   -- Element input and output operations

   procedure READ  (FILE : in FILE_TYPE;
                    FROM : in POSITIVE_COUNT);
   procedure READ  (FILE : in FILE_TYPE);

   procedure READ_EXISTING(FILE : in FILE_TYPE;
                           FROM : in POSITIVE_COUNT);
   procedure READ_EXISTING(FILE : in FILE_TYPE);

   procedure WRITE (FILE : in FILE_TYPE;
                    TO   : in POSITIVE_COUNT);
   procedure WRITE (FILE : in FILE_TYPE);

   procedure UPDATE(FILE : in FILE_TYPE);

   procedure UNLOCK   (FILE : in FILE_TYPE);

   procedure DELETE_ELEMENT(FILE : in FILE_TYPE);

   procedure SET_INDEX(FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);

   function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

   -- Exceptions

   STATUS_ERROR    : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR      : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR      : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR       : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR    : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR       : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR      : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LOCK_ERROR      : exception renames AUX_IO_EXCEPTIONS.LOCK_ERROR;
   EXISTENCE_ERROR : exception
      renames AUX_IO_EXCEPTIONS.EXISTENCE_ERROR;
private
   -- implementation-dependent
end RELATIVE_MIXED_IO;
```

References: aux_io_exceptions package 14.4, close procedure 14.2.1, count type 14.2, create procedure 14.2b.1, data_error exception 14.4, delete procedure 14.2.1, delete_element procedure 14.2b.7, device_error exception 14.4, element_size function 14.2b.2, end_error exception 14.4, end_of_buffer function 14.2b.2, end_of_file function 14.2b.7, existence_error exception 14.4, file_mode 14.1, file_type 14.1, form function 14.2.1, get_array procedure 14.2b.2, get_item procedure 14.2b.2, index 14.2b.2, index function 14.2b.7, io_exceptions package 14.4, is_open function 14.2.1, item_array 14.2b.2, item_type 14.2b.2, lock_error exception 14.4, max_element_size function 14.2b.2, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_error exception 14.4, open procedure 14.2.1, position function 14.2b.2, put_item procedure 14.2b.2, read procedure 14.2b.7, read_existing procedure 14.2b.7, reset procedure 14.2.1, set_index procedure 14.2b.7, set_position procedure 14.2b.2, status_error exception 14.4, unlock procedure 14.2b.7, update procedure 14.2b.7, use_error exception 14.4, write procedure 14.2b.7

14.2b.9 Indexed Mixed Input-Output (OpenVMS Systems Only)

The operations available for indexed mixed-type input and output are described in this section. The exception STATUS_ERROR is raised if any of these operations is attempted for a file that is not open.

See sections 14.2b.1 and 14.2b.2 for information on the file management and element input-output operations that are available for indexed mixed input and output.

This package provides a generic READ_BY_KEY procedure, which defines input for the given element type. This procedure must be instantiated with an actual type parameter for the generic parameter KEY_TYPE. It can be instantiated with or without a value for the generic parameter DEFAULT_KEY_NUMBER. The range of values for DEFAULT_KEY_NUMBER is 0 to 254. A value of 0 (the default) designates the primary key.

If the RELATION parameter to the READ_BY_KEY procedure is specified, it must have a value of EQUAL_NEXT, EQUAL, or NEXT (the default is EQUAL).

```
procedure READ(FILE : in FILE_TYPE);

generic
    type KEY_TYPE is private;
    DEFAULT_KEY_NUMBER : INTEGER := 0;
procedure READ_BY_KEY(FILE       : in  FILE_TYPE;
                      KEY        : in  KEY_TYPE;
                      KEY_NUMBER : in  INTEGER :=
                                       DEFAULT_KEY_NUMBER;
                      RELATION   : in  RELATION_TYPE := EQUAL);
```

> Operates on a file of mode IN_FILE or INOUT_FILE. In the case of the first form, returns in the file buffer the value of the next element,

according to the last specified key and relation information. In the case of the second form, returns in the file buffer the value of the element specified by the given key information; KEY gives the key value; KEY_ NUMBER designates a primary (0) or alternate key (1 to 254); and RELATION determines the kind of match to be made for the key value.

For both forms, the element read becomes the current element. In the case of the first form, the next sequential element becomes the next element, according to the most recent key and relation information. In the case of the second form, the next sequential element that matches the key and relation information specified becomes the next element. If neither the key nor the relation information changes from one READ_ BY_KEY operation to the next, the same element will continue to be read.

The exception MODE_ERROR is raised if the current mode is OUT_ FILE. The exception END_ERROR is raised if an attempt is made to read past the end of the file by the first form. The exception LOCK_ ERROR is raised if the element to be read is locked. (This error is possible only if the external file is being shared.) The exception EXISTENCE_ERROR is raised if the element does not exist. The exception KEY_ERROR is raised if the size of the given key is not a multiple of eight bits.

procedure WRITE(FILE : **in** FILE_TYPE);

Operates on a file of mode INOUT_FILE or OUT_FILE. Gives the value of the file buffer to the element whose position in the given file is specified by the key information contained within the file buffer. If the format of the associated external file is fixed and the current position in the file buffer does not indicate the end of the buffer, the rest of the file buffer is filled with zero bits before being written to the file.

The exception MODE_ERROR is raised if the current mode is IN_FILE. The exception USE_ERROR is raised if the element position in the file has already been written. The exception KEY_ERROR is raised if a key has been duplicated and if duplicates are not allowed by the external file.

procedure UPDATE(FILE : **in** FILE_TYPE);

Operates on a file of mode INOUT_FILE. Updates the current element of the given file with the contents of the file buffer. If the format of the associated external file element is fixed and the current position in the file buffer does not indicate the end of the buffer, the rest of the file buffer is filled with zero bits before being written to the file.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation or if some key specification in the file buffer violates the external file attributes defined for that key. The exception KEY_ERROR is raised if a key has been changed or duplicated and if changes or duplicates are not allowed by the external file.

procedure UNLOCK(FILE : **in** FILE_TYPE);

Operates on a file of any mode. After this operation, the current element is undefined.

procedure DELETE_ELEMENT(FILE : **in** FILE_TYPE);

Operates on a file of mode INOUT_FILE. Deletes the current element of the file.

The exception MODE_ERROR is raised if the current mode is not INOUT_FILE. The exception USE_ERROR is raised if the current element is undefined at the start of this operation.

function END_OF_FILE(FILE : **in** FILE_TYPE) **return** BOOLEAN;

Operates on a file of mode IN_FILE or INOUT_FILE. Returns TRUE if there are no more elements (according to the most recent key and relation information) starting at the next element in the file; otherwise returns FALSE.

The exception MODE_ERROR is raised if the current mode is OUT_FILE.

References: current element 14.2a, current position 14.2b, element 14.1a, element_type 14.1, end_error exception 14.4, existence_error exception 14.4, external file 14.1, file 14.1, file buffer 14.2b, file mode 14.1, file sharing 14.2a, generic actual parameter 12.3, generic formal parameter 12.3, generic procedure 12.1, indexed file 14.2a, in_file 14.1, inout_file 14.1, instantiation 12.3, key 14.2a, key_error exception 14.4, lock_error exception 14.4, locking 14.2a, match 14.2a, mixed-type file 14.1a, mode_error exception 14.4, next element 14.2a, open file 14.1, out_file 14.1, range 3.5 relation_type 14.2a, status_error exception 14.4, use_error exception 14.4

14.2b.10 Specification of the Package Indexed_Mixed_IO (OpenVMS Systems Only)

```
with IO_EXCEPTIONS;
with AUX_IO_EXCEPTIONS;
package INDEXED_MIXED_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

   type RELATION_TYPE is (EQUAL_NEXT, EQUAL, NEXT);

   function GREATER return RELATION_TYPE
      renames NEXT;
   function GREATER_EQUAL return RELATION_TYPE
      renames EQUAL_NEXT;

   type COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

   -- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE := INOUT_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING);

   -- All information about the keys must be provided in the
   -- FORM parameter of the CREATE procedure.

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE;
                    NAME : in STRING;
                    FORM : in STRING := "");

   procedure CLOSE (FILE :  in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);
   procedure RESET (FILE       : in FILE_TYPE;
                    MODE       : in FILE_MODE;
                    KEY_NUMBER : in INTEGER := 0);
   procedure RESET (FILE       : in FILE_TYPE;
                    KEY_NUMBER : in INTEGER := 0);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME   (FILE : in FILE_TYPE) return STRING;
   function FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

   -- Item input and output operations

   generic
      type ITEM_TYPE is private;
   procedure GET_ITEM(FILE : in FILE_TYPE; ITEM : out ITEM_TYPE);
```

```ada
generic
   type ITEM_TYPE is private;
   type INDEX is (<>);
   type ITEM_ARRAY is array (INDEX range <>) of ITEM_TYPE;
procedure GET_ARRAY(FILE  : in FILE_TYPE;
                    ITEMS : out ITEM_ARRAY;
                    LAST  : out INDEX);

generic
   type ITEM_TYPE is private;
procedure PUT_ITEM(FILE : in FILE_TYPE; ITEM : in ITEM_TYPE);

function END_OF_BUFFER(FILE : in FILE_TYPE) return BOOLEAN;

procedure SET_POSITION(FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);

function POSITION(FILE : in FILE_TYPE) return POSITIVE_COUNT;

function MAX_ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

function ELEMENT_SIZE(FILE : in FILE_TYPE) return COUNT;

-- Element input and output operations

procedure READ  (FILE : in FILE_TYPE);

generic
   type KEY_TYPE is private;
   DEFAULT_KEY_NUMBER : INTEGER := 0;
procedure READ_BY_KEY(FILE       : in  FILE_TYPE;
                      KEY        : in  KEY_TYPE;
                      KEY_NUMBER : in  INTEGER :=
                                          DEFAULT_KEY_NUMBER;
                      RELATION   : in  RELATION_TYPE := EQUAL);

procedure WRITE (FILE : in FILE_TYPE);

procedure UPDATE(FILE : in FILE_TYPE);

procedure UNLOCK(FILE : in FILE_TYPE);

procedure DELETE_ELEMENT(FILE : in FILE_TYPE);

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

```
      -- Exceptions

   STATUS_ERROR      : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR        : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR        : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR         : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR      : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR         : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR        : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LOCK_ERROR        : exception renames AUX_IO_EXCEPTIONS.LOCK_ERROR;
   KEY_ERROR         : exception renames AUX_IO_EXCEPTIONS.KEY_ERROR;
   EXISTENCE_ERROR : exception
      renames AUX_IO_EXCEPTIONS.EXISTENCE_ERROR;

private
   -- implementation-dependent
end INDEX_MIXED_IO;
```

References: aux_io_exceptions package 14.4, close procedure 14.2.1, count type 14.2, create procedure 14.2b.1, data_error exception 14.4, delete procedure 14.2.1, delete_element procedure 14.2b.9, device_error exception 14.4, element_size function 14.2b.2, end_error exception 14.4, end_of_buffer function 14.2b.2, end_of_file function 14.2b.9, existence_error exception 14.4, file_mode 14.1, file_type 14.1, form function 14.2.1, get_array procedure 14.2b.2, get_item procedure 14.2b.2, index 14.2b.2, io_exceptions package 14.4, is_open function 14.2.1, item_array 14.2b.2, item_type 14.2b.2, key_error exception 14.4, lock_error exception 14.4, max_element_size function 14.2b.2, mode function 14.2.1, mode_error exception 14.4, name function 14.2.1, name_error exception 14.4, open procedure 14.2.1, position function 14.2b.2, put_item procedure 14.2b.2, read procedure 14.2b.9, read_by_key procedure 14.2b.9, relation_type 14.2a, reset procedure 14.2a.1, set_position procedure 14.2b.2, status_error exception 14.4, unlock procedure 14.2b.9, update procedure 14.2b.9, use_error exception 14.4, write procedure 14.2b.9

14.3 Text Input-Output

1   This section describes the package TEXT_IO, which provides facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. The specification of the package is given below in section 14.3.10. [14]

2   The facilities for file management given above, in sections 14.2.1 and 14.2.2, are available for text input-output. In place of READ and WRITE, however, there are procedures GET and PUT that input values of suitable types from text files, and output values to them. These values are provided to the PUT procedures, and returned by the GET procedures, in a parameter ITEM. Several overloaded procedures of these names exist, for different types of ITEM. These GET procedures analyze the input sequences of characters as

---

[14] See also Appendix G, AI-00355.

lexical elements (see chapter 2) and return the corresponding values; the PUT procedures output the given values as appropriate lexical elements. Procedures GET and PUT are also available that input and output individual characters treated as character values rather than as lexical elements.

3   In addition to the procedures GET and PUT for numeric and enumeration types of ITEM that operate on text files, analogous procedures are provided that read from and write to a parameter of type STRING. These procedures perform the same analysis and composition of character sequences as their counterparts which have a file parameter.

4   For all GET and PUT procedures that operate on text files, and for many other subprograms, there are forms with and without a file parameter. Each such GET procedure operates on an input file, and each such PUT procedure operates on an output file. If no file is specified, a default input file or a default output file is used.

5   At the beginning of program execution the default input and output files are the so-called standard input file and standard output file. These files are open, have respectively the current modes IN_FILE and OUT_FILE, and are associated with two implementation-defined external files. Procedures are provided to change the current default input file and the current default output file.

On OpenVMS systems, DEC Ada has two logical file names, ADA$INPUT and ADA$OUTPUT, which can be defined to refer to the standard input and standard output files. If these logical names are not defined, then the standard input and standard output files are represented by the default system input and output logical names SYS$INPUT and SYS$OUTPUT.

On Digital UNIX systems, the DEC Ada standard input file corresponds to the Digital UNIX file descriptor 0; the standard output file corresponds to the Digital UNIX file descriptor 1.

See the DEC Ada run-time reference manuals for more information.

6   From a logical point of view, a text file is a sequence of pages, a page is a sequence of lines, and a line is a sequence of characters; the end of a line is marked by a *line terminator*; the end of a page is marked by the combination of a line terminator immediately followed by a *page terminator*; and the end of a file is marked by the combination of a line terminator immediately followed by a page terminator and then a *file terminator.* Terminators are generated during output; either by calls of procedures provided expressly for that purpose; or implicitly as part of other operations, for example, when a bounded line length, a bounded page length, or both, have been specified for a file.

7   The actual nature of terminators is not defined by the language and hence depends on the implementation. Although terminators are recognized or generated by certain of the procedures that follow, they are not necessarily implemented as characters or as sequences of characters. Whether they are characters (and if so which ones) in any particular implementation need not concern a user who neither explicitly outputs nor explicitly inputs control characters. The effect of input or output of control characters (other than horizontal tabulation) is not defined by the language.

The DEC Ada run-time reference manuals describe how line, page, and file terminators are interpreted in DEC Ada.

8   The characters of a line are numbered, starting from one; the number of a character is called its *column number.* For a line terminator, a column number is also defined: it is one more than the number of characters in the line. The lines of a page, and the pages of a file, are similarly numbered. The *current column number* is the column number of the next character or line terminator to be transferred. The *current line number* is the number of the current line. The *current page number* is the number of the current page. These numbers are values of the subtype POSITIVE_COUNT of the type COUNT (by convention, the value zero of the type COUNT is used to indicate special conditions).

```
type COUNT is range 0 .. implementation_defined;
subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
```

In DEC Ada, the integer type COUNT is **range** 0 .. INTEGER'LAST.

9   For an output file, a *maximum line length* can be specified and a *maximum page length* can be specified. If a value to be output cannot fit on the current line, for a specified maximum line length, then a new line is automatically started before the value is output; if, further, this new line cannot fit on the current page, for a specified maximum page length, then a new page is automatically started before the value is output. Functions are provided to determine the maximum line length and the maximum page length. When a file is opened with mode OUT_FILE, both values are zero: by convention, this means that the line lengths and page lengths are unbounded. (Consequently, output consists of a single line if the subprograms for explicit control of line and page structure are not used.) The constant UNBOUNDED is provided for this purpose.

10 **References** : count type 14.3.10, default current input file 14.3.2, default current output file 14.3.2, external file 14.1, file 14.1, get procedure 14.3.5, in_file 14.1, out_file 14.1, put procedure 14.3.5, read 14.2.2, sequential access 14.1, standard input file 14.3.2, standard output file 14.3.2

14.3.1 File Management

1 The only allowed file modes for text files are the modes IN_FILE and OUT_ FILE. The subprograms given in section 14.2.1 for the control of external files, and the function END_OF_FILE given in section 14.2.2 for sequential input-output, are also available for text files. There is also a version of END_OF_ FILE that refers to the current default input file. For text files, the procedures have the following additional effects:

2 • For the procedures CREATE and OPEN: After opening a file with mode OUT_FILE, the page length and line length are unbounded (both have the conventional value zero). After opening a file with mode IN_FILE or OUT_FILE, the current column, current line, and current page numbers are set to one.

3 • For the procedure CLOSE: If the file has the current mode OUT_FILE, has the effect of calling NEW_PAGE, unless the current page is already terminated; then outputs a file terminator.

4 • For the procedure RESET: If the file has the current mode OUT_FILE, has the effect of calling NEW_PAGE, unless the current page is already terminated; then outputs a file terminator. If the new file mode is OUT_FILE, the page and line lengths are unbounded. For all modes, the current column, line, and page numbers are set to one. [15]

5 The exception MODE_ERROR is raised by the procedure RESET upon an attempt to change the mode of a file that is either the current default input file, or the current default output file. [16]

6 **References** : create procedure 14.2.1, current column number 14.3, current default input file 14.3, current line number 14.3, current page number 14.3, end_of_file 14.3, external file 14.1, file 14.1, file mode 14.1, file terminator 14.3, in_file 14.1, line length 14.3, mode_error exception 14.4, open procedure 14.2.1, out_file 14.1, page length 14.3, reset procedure 14.2.1

14.3.2 Default Input and Output Files

1 The following subprograms provide for the control of the particular default files that are used when a file parameter is omitted from a GET, PUT or other operation of text input-output described below. [17]

2 **procedure** SET_INPUT(FILE : **in** FILE_TYPE);

3 Operates on a file of mode IN_FILE. Sets the current default input file to FILE.

---

[15] See also Appendix G, AI-00047 and AI-00486.
[16] See also Appendix G, AI-00048.
[17] See also Appendix G, AI-00048.

4   The exception STATUS_ERROR is raised if the given file is not open. The exception MODE_ERROR is raised if the mode of the given file is not IN_FILE.

5   **procedure** SET_OUTPUT(FILE : **in** FILE_TYPE);

6   Operates on a file of mode OUT_FILE. Sets the current default output file to FILE.

7   The exception STATUS_ERROR is raised if the given file is not open. The exception MODE_ERROR is raised if the mode of the given file is not OUT_FILE.

8   **function** STANDARD_INPUT **return** FILE_TYPE;

9   Returns the standard input file (see 14.3).

10   **function** STANDARD_OUTPUT **return** FILE_TYPE;

11   Returns the standard output file (see 14.3).

12   **function** CURRENT_INPUT **return** FILE_TYPE;

13   Returns the current default input file.

14   **function** CURRENT_OUTPUT **return** FILE_TYPE;

15   Returns the current default output file.

Note:

16   The standard input and the standard output files cannot be opened, closed, reset, or deleted, because the parameter FILE of the corresponding procedures has the mode **in out.**

17   **References** : current default file 14.3, default file 14.3, file_type 14.1, get procedure 14.3.5, mode_error exception 14.4, put procedure 14.3.5, status_error exception 14.4

14.3.3 Specification of Line and Page Lengths

1   The subprograms described in this section are concerned with the line and page structure of a file of mode OUT_FILE. They operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the current default output file. They provide for output of text with a specified maximum line length or page length. In these cases, line and page terminators are output implicitly and automatically when needed. When line and page lengths are unbounded (that is, when they have the conventional value zero), as in the case of a newly opened file, new lines and new pages are only started when explicitly called for.

2   In all cases, the exception STATUS_ERROR is raised if the file to be used is
     not open; the exception MODE_ERROR is raised if the mode of the file is not
     OUT_FILE.

3   **procedure** SET_LINE_LENGTH(FILE : **in** FILE_TYPE; TO : **in** COUNT);
     procedure SET_LINE_LENGTH(TO   : **in** COUNT);

4   Sets the maximum line length of the specified output file to the number
          of characters specified by TO. The value zero for TO specifies an
          unbounded line length.

5   The exception USE_ERROR is raised if the specified line length is
          inappropriate for the associated external file.

6   **procedure** SET_PAGE_LENGTH(FILE : **in** FILE_TYPE; TO : **in** COUNT);
     procedure SET_PAGE_LENGTH(TO   : **in** COUNT);

7   Sets the maximum page length of the specified output file to the
          number of lines specified by TO. The value zero for TO specifies an
          unbounded page length.

8   The exception USE_ERROR is raised if the specified page length is
          inappropriate for the associated external file.

9   **function** LINE_LENGTH(FILE : **in** FILE_TYPE) **return** COUNT;
     function LINE_LENGTH **return** COUNT;

10   Returns the maximum line length currently set for the specified output
          file, or zero if the line length is unbounded.

11   **function** PAGE_LENGTH(FILE : **in** FILE_TYPE) **return** COUNT;
     function PAGE_LENGTH **return** COUNT;

12   Returns the maximum page length currently set for the specified
          output file, or zero if the page length is unbounded.

13   **References** : count type 14.3, current default output file 14.3, external file 14.1, file
     14.1, file_type 14.1, line 14.3, line length 14.3, line terminator 14.3, maximum line
     length 14.3, maximum page length 14.3, mode_error exception 14.4, open file 14.1,
     out_file 14.1, page 14.3, page length 14.3, page terminator 14.3, status_error exception
     14.4, unbounded page length 14.3, use_error exception 14.4

14.3.4 Operations on Columns, Lines, and Pages

1   The subprograms described in this section provide for explicit control of line and page structure; they operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the appropriate (input or output) current default file. [18] The exception STATUS_ERROR is raised by any of these subprograms if the file to be used is not open.

2   ```
procedure NEW_LINE(FILE : in FILE_TYPE;
                    SPACING : in POSITIVE_COUNT := 1);
```

```
procedure NEW_LINE(SPACING : in POSITIVE_COUNT := 1);
```

Operates on a file of mode OUT_FILE.

3   For a SPACING of one: Outputs a line terminator and sets the current column number to one. Then increments the current line number by one, except in the case that the current line number is already greater than or equal to the maximum page length, for a bounded page length; in that case a page terminator is output, the current page number is incremented by one, and the current line number is set to one.

4   For a SPACING greater than one, the above actions are performed SPACING times.

5   The exception MODE_ERROR is raised if the mode is not OUT_FILE.

6   ```
procedure SKIP_LINE(FILE   : in FILE_TYPE;
                    SPACING : in POSITIVE_COUNT := 1);
```

```
procedure SKIP_LINE(SPACING : in POSITIVE_COUNT := 1);
```

7   Operates on a file of mode IN_FILE.

8   For a SPACING of one: Reads and discards all characters until a line terminator has been read, and then sets the current column number to one. If the line terminator is not immediately followed by a page terminator, the current line number is incremented by one. Otherwise, if the line terminator is immediately followed by a page terminator, then the page terminator is skipped, the current page number is incremented by one, and the current line number is set to one.

9   For a SPACING greater than one, the above actions are performed SPACING times.

---

[18] See also Appendix G, AI-00320.

10   The exception MODE_ERROR is raised if the mode is not IN_FILE. The exception END_ERROR is raised if an attempt is made to read a file terminator.

11   **function** END_OF_LINE(FILE : **in** FILE_TYPE) **return** BOOLEAN;
    function END_OF_LINE **return** BOOLEAN;

12   Operates on a file of mode IN_FILE. Returns TRUE if a line terminator or a file terminator is next; otherwise returns FALSE.

13   The exception MODE_ERROR is raised if the mode is not IN_FILE.

14   **procedure** NEW_PAGE(FILE : **in** FILE_TYPE);
    procedure NEW_PAGE;

15   Operates on a file of mode OUT_FILE. Outputs a line terminator if the current line is not terminated, or if the current page is empty (that is, if the current column and line numbers are both equal to one). Then outputs a page terminator, which terminates the current page. Adds one to the current page number and sets the current column and line numbers to one.

16   The exception MODE_ERROR is raised if the mode is not OUT_FILE.

17   **procedure** SKIP_PAGE(FILE: **in** FILE_TYPE);
    procedure SKIP_PAGE;

18   Operates on a file of mode IN_FILE. Reads and discards all characters and line terminators until a page terminator has been read. Then adds one to the current page number, and sets the current column and line numbers to one.

19   The exception MODE_ERROR is raised if the mode is not IN_FILE. The exception END_ERROR is raised if an attempt is made to read a file terminator.

20   **function** END_OF_PAGE(FILE : **in** FILE_TYPE) **return** BOOLEAN;
    function END_OF_PAGE **return** BOOLEAN;

21   Operates on a file of mode IN_FILE. Returns TRUE if the combination of a line terminator and a page terminator is next, or if a file terminator is next; otherwise returns FALSE.

22   The exception MODE_ERROR is raised if the mode is not IN_FILE.

23   **function** END_OF_FILE(FILE : **in** FILE_TYPE) **return** BOOLEAN;
    function END_OF_FILE **return** BOOLEAN;

24   Operates on a file of mode IN_FILE. Returns TRUE if a file terminator is next, or if the combination of a line, a page, and a file terminator is next; otherwise returns FALSE.

25   The exception MODE_ERROR is raised if the mode is not IN_FILE.

26   The following subprograms provide for the control of the current position of reading or writing in a file. In all cases, the default file is the current output file.

27
```
procedure SET_COL(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
procedure SET_COL(TO   : in POSITIVE_COUNT);
```

28   If the file mode is OUT_FILE:

29   If the value specified by TO is greater than the current column number, outputs spaces, adding one to the current column number after each space, until the current column number equals the specified value. If the value specified by TO is equal to the current column number, there is no effect. If the value specified by TO is less than the current column number, has the effect of calling NEW_LINE (with a spacing of one), then outputs (TO − 1) spaces, and sets the current column number to the specified value.

30   The exception LAYOUT_ERROR is raised if the value specified by TO exceeds LINE_LENGTH when the line length is bounded (that is, when it does not have the conventional value zero).

31   If the file mode is IN_FILE:

32   Reads (and discards) individual characters, line terminators, and page terminators, until the next character to be read has a column number that equals the value specified by TO; there is no effect if the current column number already equals this value. Each transfer of a character or terminator maintains the current column, line, and page numbers in the same way as a GET procedure (see 14.3.5). (Short lines will be skipped until a line is reached that has a character at the specified column position.)

33   The exception END_ERROR is raised if an attempt is made to read a file terminator.

34
```
procedure SET_LINE(FILE : in FILE_TYPE;
                   TO   : in POSITIVE_COUNT);

procedure SET_LINE(TO   : in POSITIVE_COUNT);
```

35   If the file mode is OUT_FILE:

<table>
<tr><td>36</td><td>If the value specified by TO is greater than the current line number, has the effect of repeatedly calling NEW_LINE (with a spacing of one), until the current line number equals the specified value. If the value specified by TO is equal to the current line number, there is no effect. If the value specified by TO is less than the current line number, has the effect of calling NEW_PAGE followed by a call of NEW_LINE with a spacing equal to (TO − 1).</td></tr>
</table>

36   If the value specified by TO is greater than the current line number, has the effect of repeatedly calling NEW_LINE (with a spacing of one), until the current line number equals the specified value. If the value specified by TO is equal to the current line number, there is no effect. If the value specified by TO is less than the current line number, has the effect of calling NEW_PAGE followed by a call of NEW_LINE with a spacing equal to (TO − 1).

37   The exception LAYOUT_ERROR is raised if the value specified by TO exceeds PAGE_LENGTH when the page length is bounded (that is, when it does not have the conventional value zero).

38   If the mode is IN_FILE:

39   Has the effect of repeatedly calling SKIP_LINE (with a spacing of one), until the current line number equals the value specified by TO; there is no effect if the current line number already equals this value. (Short pages will be skipped until a page is reached that has a line at the specified line position.)

40   The exception END_ERROR is raised if an attempt is made to read a file terminator.

41
```
function COL(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function COL return POSITIVE_COUNT;
```

42   Returns the current column number.

43   The exception LAYOUT_ERROR is raised if this number exceeds COUNT'LAST.

44
```
function LINE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function LINE return POSITIVE_COUNT;
```

45   Returns the current line number.

46   The exception LAYOUT_ERROR is raised if this number exceeds COUNT'LAST.

47
```
function PAGE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function PAGE return POSITIVE_COUNT;
```

48   Returns the current page number.

49   The exception LAYOUT_ERROR is raised if this number exceeds COUNT'LAST.

50 The column number, line number, or page number are allowed to exceed COUNT'LAST (as a consequence of the input or output of sufficiently many characters, lines, or pages). These events do not cause any exception to be raised. However, a call of COL, LINE, or PAGE raises the exception LAYOUT_ERROR if the corresponding number exceeds COUNT'LAST.

Note:

51 A page terminator is always skipped whenever the preceding line terminator is skipped. An implementation may represent the combination of these terminators by a single character, provided that it is properly recognized at input.

The interpretation of these terminators is given in the DEC Ada run-time reference manuals.

52 **References** : current column number 14.3, current default file 14.3, current line number 14.3, current page number 14.3, end_error exception 14.4, file 14.1, file terminator 14.3, get procedure 14.3.5, in_file 14.1, layout_error exception 14.4, line 14.3, line number 14.3, line terminator 14.3, maximum page length 14.3, mode_error exception 14.4, open file 14.1, page 14.3, page length 14.3, page terminator 14.3, positive count 14.3, status_error exception 14.4

14.3.5 Get and Put Procedures

1 The procedures GET and PUT for items of the types CHARACTER, STRING, numeric types, and enumeration types are described in subsequent sections. Features of these procedures that are common to most of these types are described in this section. The GET and PUT procedures for items of type CHARACTER and STRING deal with individual character values; the GET and PUT procedures for numeric and enumeration types treat the items as lexical elements.

2 All procedures GET and PUT have forms with a file parameter, written first. Where this parameter is omitted, the appropriate (input or output) current default file is understood to be specified. Each procedure GET operates on a file of mode IN_FILE. Each procedure PUT operates on a file of mode OUT_FILE.

3 All procedures GET and PUT maintain the current column, line, and page numbers of the specified file: the effect of each of these procedures upon these numbers is the resultant of the effects of individual transfers of characters and of individual output or skipping of terminators. Each transfer of a character adds one to the current column number. Each output of a line terminator sets the current column number to one and adds one to the current line number. Each output of a page terminator sets the current column and line numbers to one and adds one to the current page number. For input, each skipping of a line terminator sets the current column number to one and adds one to the current line number; each skipping of a page terminator sets the current

column and line numbers to one and adds one to the current page number. Similar considerations apply to the procedures GET_LINE, PUT_LINE, and SET_COL. [19]

4   Several GET and PUT procedures, for numeric and enumeration types, have *format* parameters which specify field lengths; these parameters are of the nonnegative subtype FIELD of the type INTEGER.

5   Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any GET procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators; a *blank* being defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

6   For a numeric type, the GET procedures have a format parameter called WIDTH. If the value given for this parameter is zero, the GET procedure proceeds in the same manner as for enumeration types, but using the syntax of numeric literals instead of that of enumeration literals. If a nonzero value is given, then exactly WIDTH characters are input, or the characters up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. The syntax used for numeric literals is an extended syntax that allows a leading sign (but no intervening blanks, or line or page terminators).

7   Any PUT procedure, for an item of a numeric or an enumeration type, outputs the value of the item as a numeric literal, identifier, or character literal, as appropriate. This is preceded by leading spaces if required by the format parameters WIDTH or FORE (as described in later sections), and then a minus sign for a negative value; for an enumeration type, the spaces follow instead of leading. The format given for a PUT procedure is overridden if it is insufficiently wide. [20]

8   Two further cases arise for PUT procedures for numeric and enumeration types, if the line length of the specified output file is bounded (that is, if it does not have the conventional value zero). If the number of characters to be output does not exceed the maximum line length, but is such that they cannot fit on the current line, starting from the current column, then (in effect) NEW_LINE is called (with a spacing of one) before output of the item. Otherwise, if the number of characters exceeds the maximum line length, then the exception LAYOUT_ERROR is raised and no characters are output.

---

[19] See also Appendix G, AI-00320.
[20] See also Appendix G, AI-00243.

9   The exception STATUS_ERROR is raised by any of the procedures GET, GET_LINE, PUT, and PUT_LINE if the file to be used is not open. The exception MODE_ERROR is raised by the procedures GET and GET_LINE if the mode of the file to be used is not IN_FILE; and by the procedures PUT and PUT_LINE, if the mode is not OUT_FILE.

10 The exception END_ERROR is raised by a GET procedure if an attempt is made to skip a file terminator. The exception DATA_ERROR is raised by a GET procedure if the sequence finally input is not a lexical element corresponding to the type, in particular if no characters were input; for this test, leading blanks are ignored; for an item of a numeric type, when a sign is input, this rule applies to the succeeding numeric literal. The exception LAYOUT_ERROR is raised by a PUT procedure that outputs to a parameter of type STRING, if the length of the actual string is insufficient for the output of the item.

11 **Examples:**

12 In the examples, here and in sections 14.3.7 and 14.3.8, the string quotes and the lower case letter b are not transferred: they are shown only to reveal the layout and spaces.

```
N : INTEGER;
   ...
GET(N);

-- Characters at input    Sequence input    Value of N
--      bb-12535b          -12535            -12535
--      bb12_535E1b         12_535E1          125350
--      bb12_535E;          12_535E          (none)
--                                           DATA_ERROR raised
```

13 **Example of overridden width parameter:**
```
PUT(ITEM => -23, WIDTH => 2);  --  "-23"
```

14 **References** : blank 14.3.9, column number 14.3, current default file 14.3, data_error exception 14.4, end_error exception 14.4, file 14.1, fore 14.3.8, get procedure 14.3.6 14.3.7 14.3.8 14.3.9, in_file 14.1, layout_error exception 14.4, line number 14.1, line terminator 14.1, maximum line length 14.3, mode 14.1, mode_error exception 14.4, new_file procedure 14.3.4, out_file 14.1, page number 14.1, page terminator 14.1, put procedure 14.3.6 14.3.7 14.3.8 14.3.9, skipping 14.3.7 14.3.8 14.3.9, status_error exception 14.4, width 14.3.5 14.3.7 14.3.9

14.3.6 Input-Output of Characters and Strings

1   For an item of type CHARACTER the following procedures are provided:

2   **procedure** GET(FILE : **in** FILE_TYPE;
                      ITEM : **out** CHARACTER);

     procedure GET(ITEM : **out** CHARACTER);

3   After skipping any line terminators and any page terminators, reads
               the next character from the specified input file and returns the value of
               this character in the **out** parameter ITEM.

4   The exception END_ERROR is raised if an attempt is made to skip a
               file terminator.

5   **procedure** PUT(FILE : **in** FILE_TYPE; ITEM : **in** CHARACTER);
     procedure PUT(ITEM : **in** CHARACTER);

6   If the line length of the specified output file is bounded (that is, does
               not have the conventional value zero), and the current column number
               exceeds it, has the effect of calling NEW_LINE with a spacing of one.
               Then, or otherwise, outputs the given character to the file.

7   For an item of type STRING the following procedures are provided:

8   **procedure** GET(FILE : **in** FILE_TYPE; ITEM : **out** STRING);
     procedure GET(ITEM : **out** STRING);

9   Determines the length of the given string and attempts that number of
               GET operations for successive characters of the string (in particular, no
               operation is performed if the string is null).

10   **procedure** PUT(FILE : **in** FILE_TYPE; ITEM : **in** STRING);
     procedure PUT(ITEM : **in** STRING);

11   Determines the length of the given string and attempts that number of
               PUT operations for successive characters of the string (in particular, no
               operation is performed if the string is null).

12   **procedure** GET_LINE(FILE : **in** FILE_TYPE;
                           ITEM : **out** STRING;
                           LAST : **out** NATURAL);

     procedure GET_LINE(ITEM : **out** STRING;
                           LAST : **out** NATURAL);

13   Replaces successive characters of the specified string by successive
               characters read from the specified input file. Reading stops if the end
               of the line is met, in which case the procedure SKIP_LINE is then

called (in effect) with a spacing of one; reading also stops if the end of the string is met. Characters not replaced are left undefined. [21]

14   If characters are read, returns in LAST the index value such that ITEM(LAST) is the last character replaced (the index of the first character replaced is ITEM'FIRST). If no characters are read, returns in LAST an index value that is one less than ITEM'FIRST.

15   The exception END_ERROR is raised if an attempt is made to skip a file terminator.

16   **procedure** PUT_LINE(FILE : **in** FILE_TYPE; ITEM : **in** STRING);
procedure PUT_LINE(ITEM : **in** STRING);

17   Calls the procedure PUT for the given string, and then the procedure NEW_LINE with a spacing of one.

Notes:

18   In a literal string parameter of PUT, the enclosing string bracket characters are not output. Each doubled string bracket character in the enclosed string is output as a single string bracket character, as a consequence of the rule for string literals (see 2.6).

19   A string read by GET or written by PUT can extend over several lines.

20   **References** : current column number 14.3, end_error exception 14.4, file 14.1, file terminator 14.3, get procedure 14.3.5, line 14.3, line length 14.3, new_line procedure 14.3.4, page terminator 14.3, put procedure 14.3.4, skipping 14.3.5

14.3.7 Input-Output for Integer Types

1   The following procedures are defined in the generic package INTEGER_IO. This must be instantiated for the appropriate integer type (indicated by NUM in the specification).

2   Values are output as decimal or based literals, without underline characters or exponent, and preceded by a minus sign if negative. The format (which includes any leading spaces and minus sign) can be specified by an optional field width parameter. Values of widths of fields in output formats are of the nonnegative integer subtype FIELD. Values of bases are of the integer subtype NUMBER_BASE.

subtype NUMBER_BASE **is** INTEGER **range** 2 .. 16;

---

[21] See also Appendix G, AI-00050 and AI-00172.

3   The default field width and base to be used by output procedures are defined by
     the following variables that are declared in the generic package INTEGER_IO:

```
DEFAULT_WIDTH : FIELD := NUM'WIDTH;
DEFAULT_BASE  : NUMBER_BASE := 10;
```

4   The following procedures are provided:

5   **procedure** GET(FILE : **in** FILE_TYPE;
                       ITEM : **out** NUM;
                       WIDTH : **in** FIELD := 0);

     procedure GET(ITEM : **out** NUM; WIDTH : **in** FIELD := 0);

6   If the value of the parameter WIDTH is zero, skips any leading blanks,
                     line terminators, or page terminators, then reads a plus or a minus
                     sign if present, then reads according to the syntax of an integer
                     literal (which may be a based literal). If a nonzero value of WIDTH is
                     supplied, then exactly WIDTH characters are input, or the characters
                     (possibly none) up to a line terminator, whichever comes first; any
                     skipped leading blanks are included in the count. [22]

7   Returns, in the parameter ITEM, the value of type NUM that
                     corresponds to the sequence input.

8   The exception DATA_ERROR is raised if the sequence input does not
                     have the required syntax or if the value obtained is not of the subtype
                     NUM.

9   **procedure** PUT(FILE : **in** FILE_TYPE;
                       ITEM : **in** NUM;
                       WIDTH : **in** FIELD := DEFAULT_WIDTH;
                       BASE : **in** NUMBER_BASE := DEFAULT_BASE);

     procedure PUT(ITEM : **in** NUM;
                       WIDTH : **in** FIELD := DEFAULT_WIDTH;
                       BASE : **in** NUMBER_BASE := DEFAULT_BASE);

10   Outputs the value of the parameter ITEM as an integer literal, with no
                     underlines, no exponent, and no leading zeros (but a single zero for the
                     value zero), and a preceding minus sign for a negative value.

11   If the resulting sequence of characters to be output has fewer than
                     WIDTH characters, then leading spaces are first output to make up the
                     difference.

---

[22] See also Appendix G, AI-00051 and AI-00307.

12   Uses the syntax for decimal literal if the parameter BASE has the
        value ten (either explicitly or through DEFAULT_BASE); otherwise,
        uses the syntax for based literal, with any letters in upper case.

13 **procedure** GET(FROM : **in** STRING;
                 ITEM : **out** NUM;
                 LAST : **out** POSITIVE);

14   Reads an integer value from the beginning of the given string, following
        the same rules as the GET procedure that reads an integer value from
        a file, but treating the end of the string as a file terminator. Returns,
        in the parameter ITEM, the value of type NUM that corresponds
        to the sequence input. Returns in LAST the index value such that
        FROM(LAST) is the last character read. [23]

15   The exception DATA_ERROR is raised if the sequence input does not
        have the required syntax or if the value obtained is not of the subtype
        NUM.

16 **procedure** PUT(TO   : **out** STRING;
                 ITEM : **in** NUM;
                 BASE : **in** NUMBER_BASE := DEFAULT_BASE);

17   Outputs the value of the parameter ITEM to the given string, following
        the same rule as for output to a file, using the length of the given string
        as the value for WIDTH.

18 **Examples:**
    package INT_IO **is new** INTEGER_IO(SMALL_INT); **use** INT_IO;
    -- default format used at instantiation,
    -- DEFAULT_WIDTH = 4, DEFAULT_BASE = 10

    PUT(126);   -- "b126"
    PUT(-126, 7);   -- "bbb-126"
    PUT(126, WIDTH => 13, BASE => 2);   -- "bbb2#1111110#"

19 **References** : based literal 2.4.2, blank 14.3.5, data_error exception 14.4, decimal
    literal 2.4.1, field subtype 14.3.5, file_type 14.1, get procedure 14.3.5, integer_io
    package 14.3.10, integer literal 2.4, layout_error exception 14.4, line terminator 14.3,
    put procedure 14.3.5, skipping 14.3.5, width 14.3.5

---

[23] See also Appendix G, AI-00307.

14.3.8 Input-Output for Real Types

1 The following procedures are defined in the generic packages FLOAT_IO and FIXED_IO, which must be instantiated for the appropriate floating point or fixed point type respectively (indicated by NUM in the specifications).

2 Values are output as decimal literals without underline characters. The format of each value output consists of a FORE field, a decimal point, an AFT field, and (if a nonzero EXP parameter is supplied) the letter E and an EXP field. The two possible formats thus correspond to:

```
FORE  .  AFT
```

3 and to:

```
FORE  .  AFT  E  EXP
```

4 without any spaces between these fields. The FORE field may include leading spaces, and a minus sign for negative values. The AFT field includes only decimal digits (possibly with trailing zeros). The EXP field includes the sign (plus or minus) and the exponent (possibly with leading zeros).

5 For floating point types, the default lengths of these fields are defined by the following variables that are declared in the generic package FLOAT_IO:

```
DEFAULT_FORE : FIELD := 2;
DEFAULT_AFT  : FIELD := NUM'DIGITS-1;
DEFAULT_EXP  : FIELD := 3;
```

6 For fixed point types, the default lengths of these fields are defined by the following variables that are declared in the generic package FIXED_IO:

```
DEFAULT_FORE : FIELD := NUM'FORE;
DEFAULT_AFT  : FIELD := NUM'AFT;
DEFAULT_EXP  : FIELD := 0;
```

7 The following procedures are provided:

8 **procedure** GET(FILE : **in** FILE_TYPE;
                    ITEM : **out** NUM;
                    WIDTH : **in** FIELD := 0);

   procedure GET(ITEM : **out** NUM;
                    WIDTH : **in** FIELD := 0);

9 If the value of the parameter WIDTH is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus or a minus sign if present, then reads according to the syntax of a real literal (which may be a based literal). If a nonzero value of WIDTH is supplied, then exactly WIDTH characters are input, or the characters (possibly none)

up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. [24]

10   Returns, in the parameter ITEM, the value of type NUM that corresponds to the sequence input.

11   The exception DATA_ERROR is raised if the sequence input does not have the required syntax or if the value obtained is not of the subtype NUM.

12
```
procedure PUT(FILE : in FILE_TYPE;
              ITEM : in NUM;
              FORE : in FIELD := DEFAULT_FORE;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP);

procedure PUT(ITEM : in NUM;
              FORE : in FIELD := DEFAULT_FORE;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP);
```

13   Outputs the value of the parameter ITEM as a decimal literal with the format defined by FORE, AFT and EXP. If the value is negative, a minus sign is included in the integer part. If EXP has the value zero, then the integer part to be output has as many digits as are needed to represent the integer part of the value of ITEM, overriding FORE if necessary, or consists of the digit zero if the value of ITEM has no integer part.

14   If EXP has a value greater than zero, then the integer part to be output has a single digit, which is nonzero except for the value 0.0 of ITEM.

15   In both cases, however, if the integer part to be output has fewer than FORE characters, including any minus sign, then leading spaces are first output to make up the difference. The number of digits of the fractional part is given by AFT, or is one if AFT equals zero. The value is rounded; a value of exactly one half in the last place may be rounded either up or down.

16   If EXP has the value zero, there is no exponent part. If EXP has a value greater than zero, then the exponent part to be output has as many digits as are needed to represent the exponent part of the value of ITEM (for which a single digit integer part is used), and includes an initial sign (plus or minus). If the exponent part to be output has fewer than EXP characters, including the sign, then leading zeros precede

---

[24] See also Appendix G, AI-00307.

the digits, to make up the difference. For the value 0.0 of ITEM, the exponent has the value zero.

17 **procedure** GET(FROM : **in** STRING;
                  ITEM : **out** NUM;
                  LAST : **out** POSITIVE);

18 Reads a real value from the beginning of the given string, following the same rule as the GET procedure that reads a real value from a file, but treating the end of the string as a file terminator. Returns, in the parameter ITEM, the value of type NUM that corresponds to the sequence input. Returns in LAST the index value such that FROM(LAST) is the last character read. [25]

19 The exception DATA_ERROR is raised if the sequence input does not have the required syntax, or if the value obtained is not of the subtype NUM.

20 **procedure** PUT(TO   : **out** STRING;
                 ITEM : **in** NUM;
                 AFT : **in** FIELD := DEFAULT_AFT;
                 EXP : **in** FIELD := DEFAULT_EXP);

    26

21 Outputs the value of the parameter ITEM to the given string, following the same rule as for output to a file, using a value for FORE such that the sequence of characters output exactly fills the string, including any leading spaces.

22 **Examples:**

```
package REAL_IO is new FLOAT_IO(REAL); use REAL_IO;
-- default format used at instantiation, DEFAULT_EXP = 3

X : REAL := -123.4567;  --  digits 8     (see 3.5.7)

PUT(X);                    -- default format "-1.2345670E+02"
PUT(X, FORE => 5, AFT => 3, EXP => 2);     --   "bbb-1.235E+2"
PUT(X, 5, 3, 0);                           --   "b-123.457"
```

---

[25] See also Appendix G, AI-00307.

[26] Specification corrected according to AI-00215; see Appendix Appendix G.

Note:

23 For an item with a positive value, if output to a string exactly fills the string without leading spaces, then output of the corresponding negative value will raise LAYOUT_ERROR.

24 **References** : aft attribute 3.5.10, based literal 2.4.2, blank 14.3.5, data_error exception 14.3.5, decimal literal 2.4.1, field subtype 14.3.5, file_type 14.1, fixed_io package 14.3.10, floating_io package 14.3.10, fore attribute 3.5.10, get procedure 14.3.5, layout_error 14.3.5, line terminator 14.3.5, put procedure 14.3.5, real literal 2.4, skipping 14.3.5, width 14.3.5

14.3.9 Input-Output for Enumeration Types

1 The following procedures are defined in the generic package ENUMERATION_IO, which must be instantiated for the appropriate enumeration type (indicated by ENUM in the specification).

2 Values are output using either upper or lower case letters for identifiers. This is specified by the parameter SET, which is of the enumeration type TYPE_SET.

```
type TYPE_SET is (LOWER_CASE, UPPER_CASE);
```

3 The format (which includes any trailing spaces) can be specified by an optional field width parameter. The default field width and letter case are defined by the following variables that are declared in the generic package ENUMERATION_IO:

```
DEFAULT_WIDTH   : FIELD := 0;
DEFAULT_SETTING : TYPE_SET := UPPER_CASE;
```

4 The following procedures are provided:

5 ```
procedure GET(FILE : in FILE_TYPE; ITEM : out ENUM);
procedure GET(ITEM : out ENUM);
```

6 After skipping any leading blanks, line terminators, or page terminators, reads an identifier according to the syntax of this lexical element (lower and upper case being considered equivalent), or a character literal according to the syntax of this lexical element (including the apostrophes). Returns, in the parameter ITEM, the value of type ENUM that corresponds to the sequence input. [27]

---

[27] See also Appendix G, AI-00239, AI-00307, and AI-00316.

7      The exception DATA_ERROR is raised if the sequence input does not
       have the required syntax, or if the identifier or character literal does
       not correspond to a value of the subtype ENUM.

8    **procedure** PUT(FILE  : **in** FILE_TYPE;
                   ITEM  : **in** ENUM;
                   WIDTH : **in** FIELD := DEFAULT_WIDTH;
                   SET   : **in** TYPE_SET := DEFAULT_SETTING);

     **procedure** PUT(ITEM  : **in** ENUM;
                   WIDTH : **in** FIELD := DEFAULT_WIDTH;
                   SET   : **in** TYPE_SET := DEFAULT_SETTING);

9      Outputs the value of the parameter ITEM as an enumeration literal
       (either an identifier or a character literal).  The optional parameter
       SET indicates whether lower case or upper case is used for identifiers;
       it has no effect for character literals.  If the sequence of characters
       produced has fewer than WIDTH characters, then trailing spaces are
       finally output to make up the difference. [28]

10   **procedure** GET(FROM : **in** STRING;
                   ITEM : **out** ENUM;
                   LAST : **out** POSITIVE);

11     Reads an enumeration value from the beginning of the given
       string, following the same rule as the GET procedure that reads
       an enumeration value from a file, but treating the end of the string as
       a file terminator.  Returns, in the parameter ITEM, the value of type
       ENUM that corresponds to the sequence input.  Returns in LAST the
       index value such that FROM(LAST) is the last character read. [29]

12     The exception DATA_ERROR is raised if the sequence input does not
       have the required syntax, or if the identifier or character literal does
       not correspond to a value of the subtype ENUM.

13   **procedure** PUT(TO   : **out** STRING;
                   ITEM : **in** ENUM;
                   SET  : **in** TYPE_SET := DEFAULT_SETTING);

14     Outputs the value of the parameter ITEM to the given string, following
       the same rule as for output to a file, using the length of the given string
       as the value for WIDTH.

15   Although the specification of the package ENUMERATION_IO would allow
     instantiation for an integer type, this is not the intended purpose of this
     generic package, and the effect of such instantiations is not defined by the
     language.

---

[28] See also Appendix G, AI-00239
[29] See also Appendix G, AI-00307.

**Notes:**

16 There is a difference between PUT defined for characters, and for enumeration values. Thus

```
TEXT_IO.PUT('A');  --  outputs the character A

package CHAR_IO is new TEXT_IO.ENUMERATION_IO(CHARACTER);
CHAR_IO.PUT('A');  --  outputs the character 'A',
                   --  between single quotes
```

17 The type BOOLEAN is an enumeration type, hence ENUMERATION_IO can be instantiated for this type.

When a control character is input or output using one of the GET or PUT operations resulting from an instantiation of the package ENUMERATION_IO for the type CHARACTER, the text input (for GET) or output (for PUT) is the two- or three-character mnemonic for the control character, as shown (in italics) in Annex C. For example:

```
CHAR_IO.PUT(ASCII.NUL); -- outputs the characters NUL
```

For the analogous operation CHAR_IO.GET, the input value would be NUL.

18 **References** : blank 14.3.5, data_error 14.3.5, enumeration_io package 14.3.10, field subtype 14.3.5, file_type 14.1, get procedure 14.3.5, line terminator 14.3.5, put procedure 14.3.5, skipping 14.3.5, width 14.3.5

character type 3.5.2, instantiation 12.3

14.3.10 Specification of the Package Text_IO

```
1    with IO_EXCEPTIONS;
     package TEXT_IO is

        type FILE_TYPE is limited private;

        type FILE_MODE is (IN_FILE, OUT_FILE);

        type COUNT is range 0 .. implementation_defined;
        subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
        UNBOUNDED : constant COUNT := 0; -- line and page length

        subtype FIELD       is INTEGER range 0 .. implementation_defined;
        subtype NUMBER_BASE is INTEGER range 2 .. 16;

        type TYPE_SET is (LOWER_CASE, UPPER_CASE);

        -- File Management

        procedure CREATE (FILE : in out FILE_TYPE;
                          MODE : in FILE_MODE := OUT_FILE;
                          NAME : in STRING    := "";
                          FORM : in STRING    := "");
```

```
procedure OPEN    (FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE;
                   NAME : in STRING;
                   FORM : in STRING := "");
procedure CLOSE  (FILE : in out FILE_TYPE);
procedure DELETE (FILE : in out FILE_TYPE);
procedure RESET  (FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE);
procedure RESET  (FILE : in out FILE_TYPE);

function  MODE   (FILE : in FILE_TYPE) return FILE_MODE;
function  NAME   (FILE : in FILE_TYPE) return STRING;
function  FORM   (FILE : in FILE_TYPE) return STRING;

function  IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- Control of default input and output files

procedure SET_INPUT (FILE : in FILE_TYPE);
procedure SET_OUTPUT(FILE : in FILE_TYPE);

function  STANDARD_INPUT  return FILE_TYPE;
function  STANDARD_OUTPUT return FILE_TYPE;

function  CURRENT_INPUT   return FILE_TYPE;
function  CURRENT_OUTPUT  return FILE_TYPE;

-- Specification of line and page lengths

procedure SET_LINE_LENGTH(FILE : in FILE_TYPE;
                          TO   : in COUNT);
procedure SET_LINE_LENGTH(TO   : in COUNT);

procedure SET_PAGE_LENGTH(FILE : in FILE_TYPE;
                          TO   : in COUNT);
procedure SET_PAGE_LENGTH(TO   : in COUNT);

function  LINE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function  LINE_LENGTH return COUNT;

function  PAGE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function  PAGE_LENGTH return COUNT;

-- Column, Line, and Page Control

procedure NEW_LINE   (FILE    : in FILE_TYPE;
                      SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE   (SPACING : in POSITIVE_COUNT := 1);

procedure SKIP_LINE  (FILE    : in FILE_TYPE;
                      SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE  (SPACING : in POSITIVE_COUNT := 1);

function  END_OF_LINE(FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_LINE return BOOLEAN;
```

```
procedure NEW_PAGE   (FILE : in FILE_TYPE);
procedure NEW_PAGE;

procedure SKIP_PAGE  (FILE : in FILE_TYPE);
procedure SKIP_PAGE;

function  END_OF_PAGE(FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_PAGE return BOOLEAN;

function  END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_FILE return BOOLEAN;

procedure SET_COL (FILE : in FILE_TYPE;
                   TO   : in POSITIVE_COUNT);
procedure SET_COL (TO   : in POSITIVE_COUNT);

procedure SET_LINE(FILE : in FILE_TYPE;
                   TO   : in POSITIVE_COUNT);
procedure SET_LINE(TO   : in POSITIVE_COUNT);

function COL (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function COL  return POSITIVE_COUNT;

function LINE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function LINE return POSITIVE_COUNT;

function PAGE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function PAGE return POSITIVE_COUNT;

-- Character Input-Output

procedure GET(FILE : in  FILE_TYPE;
              ITEM : out CHARACTER);
procedure GET(ITEM : out CHARACTER);
procedure PUT(FILE : in  FILE_TYPE;
              ITEM : in CHARACTER);
procedure PUT(ITEM : in  CHARACTER);

-- String Input-Output

procedure GET(FILE : in  FILE_TYPE;
              ITEM : out STRING);
procedure GET(ITEM : out STRING);
procedure PUT(FILE : in  FILE_TYPE;
              ITEM : in STRING);
procedure PUT(ITEM : in  STRING);

procedure GET_LINE(FILE : in  FILE_TYPE;
                   ITEM : out STRING;
                   LAST : out NATURAL);
procedure GET_LINE(ITEM : out STRING;
                   LAST : out NATURAL);
procedure PUT_LINE(FILE : in  FILE_TYPE;
                   ITEM : in STRING);
procedure PUT_LINE(ITEM : in  STRING);

-- Generic package for Input-Output of Integer Types
```

```
generic
   type NUM is range <>;
package INTEGER_IO is

   DEFAULT_WIDTH : FIELD := NUM'WIDTH;
   DEFAULT_BASE  : NUMBER_BASE := 10;

   procedure GET(FILE  : in  FILE_TYPE;
                 ITEM  : out NUM;
                 WIDTH : in FIELD := 0);
   procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);

   procedure PUT(FILE  : in FILE_TYPE;
                 ITEM  : in NUM;
                 WIDTH : in FIELD := DEFAULT_WIDTH;
                 BASE  : in NUMBER_BASE := DEFAULT_BASE);
   procedure PUT(ITEM  : in NUM;
                 WIDTH : in FIELD := DEFAULT_WIDTH;
                 BASE  : in NUMBER_BASE := DEFAULT_BASE);

   procedure GET(FROM : in  STRING;
                 ITEM : out NUM;
                 LAST : out POSITIVE);
   procedure PUT(TO   : out STRING;
                 ITEM : in NUM;
                 BASE : in NUMBER_BASE := DEFAULT_BASE);

end INTEGER_IO;

-- Generic packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO is

   DEFAULT_FORE : FIELD := 2;
   DEFAULT_AFT  : FIELD := NUM'DIGITS-1;
   DEFAULT_EXP  : FIELD := 3;

   procedure GET(FILE  : in FILE_TYPE;
                 ITEM  : out NUM;
                 WIDTH : in FIELD := 0);
   procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);

   procedure PUT(FILE : in FILE_TYPE;
                 ITEM : in NUM;
                 FORE : in FIELD := DEFAULT_FORE;
                 AFT  : in FIELD := DEFAULT_AFT;
                 EXP  : in FIELD := DEFAULT_EXP);
   procedure PUT(ITEM : in NUM;
                 FORE : in FIELD := DEFAULT_FORE;
                 AFT  : in FIELD := DEFAULT_AFT;
                 EXP  : in FIELD := DEFAULT_EXP);
```

```ada
      procedure GET(FROM  : in STRING;
                    ITEM  : out NUM;
                    LAST  : out POSITIVE);
      procedure PUT(TO    : out STRING;
                    ITEM  : in NUM;
                    AFT   : in FIELD := DEFAULT_AFT;
                    EXP   : in FIELD := DEFAULT_EXP);
   end FLOAT_IO;

   generic
      type NUM is delta <>;
   package FIXED_IO is

      DEFAULT_FORE : FIELD := NUM'FORE;
      DEFAULT_AFT  : FIELD := NUM'AFT;
      DEFAULT_EXP  : FIELD := 0;

      procedure GET(FILE  : in FILE_TYPE;
                    ITEM  : out NUM;
                    WIDTH : in FIELD := 0);
      procedure GET(ITEM  : out NUM; WIDTH : in FIELD := 0);

      procedure PUT(FILE  : in FILE_TYPE;
                    ITEM  : in NUM;
                    FORE  : in FIELD := DEFAULT_FORE;
                    AFT   : in FIELD := DEFAULT_AFT;
                    EXP   : in FIELD := DEFAULT_EXP);
      procedure PUT(ITEM  : in NUM;
                    FORE  : in FIELD := DEFAULT_FORE;
                    AFT   : in FIELD := DEFAULT_AFT;
                    EXP   : in FIELD := DEFAULT_EXP);

      procedure GET(FROM : in  STRING;
                    ITEM : out NUM;
                    LAST : out POSITIVE);
      procedure PUT(TO   : out STRING;
                    ITEM : in NUM;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);

   end FIXED_IO;

   -- Generic package for Input-Output of Enumeration Types

   generic
      type ENUM is (<>);
   package ENUMERATION_IO is

      DEFAULT_WIDTH   : FIELD := 0;
      DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

      procedure GET(FILE  : in FILE_TYPE; ITEM : out ENUM);
      procedure GET(ITEM  : out ENUM);
```

```
       procedure PUT(FILE  : in FILE_TYPE;
                     ITEM  : in ENUM;
                     WIDTH : in FIELD    := DEFAULT_WIDTH;
                     SET   : in TYPE_SET := DEFAULT_SETTING);
       procedure PUT(ITEM  : in ENUM;
                     WIDTH : in FIELD    := DEFAULT_WIDTH;
                     SET   : in TYPE_SET := DEFAULT_SETTING);

       procedure GET(FROM : in  STRING;
                     ITEM : out ENUM;
                     LAST : out POSITIVE);
       procedure PUT(TO   : out STRING;
                     ITEM : in  ENUM;
                     SET  : in  TYPE_SET := DEFAULT_SETTING);
   end ENUMERATION_IO;

   -- Exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private
   -- implementation-dependent
end TEXT_IO;
```

14.4 Exceptions in Input-Output

1   The following exceptions can be raised by input-output operations. They
     are declared in the package IO_EXCEPTIONS, defined in section 14.5;
     this package is named in the context clause for each of the three input-
     output packages. Only outline descriptions are given of the conditions under
     which NAME_ERROR, USE_ERROR, and DEVICE_ERROR are raised;
     for full details see Appendix F. If more than one error condition exists, the
     corresponding exception that appears earliest in the following list is the one
     that is raised.

     In DEC Ada, the exception DEVICE_ERROR is never raised. Device-
     related errors raise the exception USE_ERROR. The input-output operation
     descriptions in the previous sections give the conditions under which the
     exceptions NAME_ERROR and USE_ERROR are raised. Appendix F
     summarizes these conditions.

2   The exception STATUS_ERROR is raised by an attempt to operate upon a file
     that is not open, and by an attempt to open a file that is already open.

3   The exception MODE_ERROR is raised by an attempt to read from, or test for
     the end of, a file whose current mode is OUT_FILE, and also by an attempt
     to write to a file whose current mode is IN_FILE. In the case of TEXT_IO,
     the exception MODE_ERROR is also raised by specifying a file whose current
     mode is OUT_FILE in a call of SET_INPUT, SKIP_LINE, END_OF_LINE,
     SKIP_PAGE, or END_OF_PAGE; and by specifying a file whose current mode
     is IN_FILE in a call of SET_OUTPUT, SET_LINE_LENGTH, SET_PAGE_
     LENGTH, LINE_LENGTH, PAGE_LENGTH, NEW_LINE, or NEW_PAGE.

4   The exception NAME_ERROR is raised by a call of CREATE or OPEN if the
     string given for the parameter NAME does not allow the identification of an
     external file. For example, this exception is raised if the string is improper, or,
     alternatively, if either none or more than one external file corresponds to the
     string. [30]

5   The exception USE_ERROR is raised if an operation is attempted that is not
     possible for reasons that depend on characteristics of the external file. For
     example, this exception is raised by the procedure CREATE, among other
     circumstances, if the given mode is OUT_FILE but the form specifies an input
     only device, if the parameter FORM specifies invalid access rights, or if an
     external file with the given name already exists and overwriting is not allowed.
     [31]

6   The exception DEVICE_ERROR is raised if an input-output operation cannot
     be completed because of a malfunction of the underlying system.

     The exception DEVICE_ERROR is never raised in DEC Ada. Any device-
     related errors will raise the exception USE_ERROR.

7   The exception END_ERROR is raised by an attempt to skip (read past) the end
     of a file.

8   The exception DATA_ERROR may be raised by the procedure READ if the
     element read cannot be interpreted as a value of the required type. This
     exception is also raised by a procedure GET (defined in the package TEXT_IO)
     if the input character sequence fails to satisfy the required syntax, or if the
     value input does not belong to the range of the required type or subtype.

9   The exception LAYOUT_ERROR is raised (in text input-output) by COL,
     LINE, or PAGE if the value returned exceeds COUNT'LAST. The exception
     LAYOUT_ERROR is also raised on output by an attempt to set column or line
     numbers in excess of specified maximum line or page lengths, respectively
     (excluding the unbounded cases). It is also raised by an attempt to PUT too
     many characters to a string.

---

[30] See also Appendix G, AI-00332.
[31] See also Appendix G, AI-00332.

The exception LAYOUT_ERROR can also be raised in DEC Ada by the GET_ITEM and PUT_ITEM procedures in the mixed-type packages. LAYOUT_ERROR is raised for GET_ITEM if no more items can be read from the file buffer. It is raised for PUT_ITEM if the current position exceeds the file buffer size.

On OpenVMS systems, the following exceptions can also be raised by the DEC Ada input-output operations described in sections 14.2a.2, 14.2a.4, 14.2b.7, and 14.2b.9. These exceptions are declared in the DEC Ada package AUX_IO_EXCEPTIONS (defined in 14.5a), which is named in the context clause for the packages RELATIVE_IO, RELATIVE_MIXED_IO, INDEXED_IO, and INDEXED_MIXED_IO:

- The exception LOCK_ERROR is raised by the READ or READ_EXISTING procedures if the result is a locked record error in a relative or indexed file. See section 14.1 for a general explanation of record locking. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for a more detailed explanation.

- The exception EXISTENCE_ERROR is raised by the READ or READ_EXISTING procedures if the element to be read cannot be found in a relative or indexed file.

- The exception KEY_ERROR is raised in an indexed file if the key has been changed or duplicated and changes or duplicates are not permitted. The exception KEY_ERROR is also raised by the READ_BY_KEY procedures if the size of the given key is not a multiple of eight bits.

10 **References** : col function 14.3.4, create procedure 14.2.1, end_of_line function 14.3.4, end_of_page function 14.3.4, external file 14.1, file 14.1, form string 14.1, get procedure 14.3.5, in_file 14.1, io_exceptions package 14.5, line function 14.3.4, line_length function 14.3.4, name string 14.1, new_line procedure 14.3.4, new_page procedure 14.3.4, open procedure 14.2.1, out_file 14.1, page function 14.3.4, page_length function 14.3.4, put procedure 14.3.5, read procedure 14.2.2 14.2.3, set_input procedure 14.3.2, set_line_length 14.3.3, set_page_length 14.3.3, set_output 14.3.2, skip_line procedure 14.3.4, skip_page procedure 14.3.4, text_io package 14.3

indexed_io package 14.2a 14.2a.4, indexed_mixed_io package 14.2a 14.2b 14.2b.9, key 14.2a, locking 14.2a, read procedure 14.2a.2 14.2a.4 14.2b.7 14.2b.9, read_existing procedure 14.2a.2 14.2b.7, relative_io package 14.2a 14.2a.2, relative_mixed_io package 14.2a 14.2b 14.2b.7

14.5 Specification of the Package IO_Exceptions

1   This package defines the exceptions needed by the packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

2   **package** IO_EXCEPTIONS **is**

```
    STATUS_ERROR : exception;
    MODE_ERROR   : exception;
    NAME_ERROR   : exception;
    USE_ERROR    : exception;
    DEVICE_ERROR : exception;
    END_ERROR    : exception;
    DATA_ERROR   : exception;
    LAYOUT_ERROR : exception;
```

end IO_EXCEPTIONS;

14.5a Specification of the Package Aux_IO_Exceptions (OpenVMS Systems Only)

This package defines the exceptions needed by the DEC Ada packages RELATIVE_IO, INDEXED_IO, RELATIVE_MIXED_IO, and INDEXED_MIXED_IO.

```
package AUX_IO_EXCEPTIONS is

    LOCK_ERROR      : exception;
    EXISTENCE_ERROR : exception;
    KEY_ERROR       : exception;

end AUX_IO_EXCEPTIONS;
```

14.6 Low Level Input-Output

DEC Ada does not provide the low level input-output package described in this section.

1   A low level input-output operation is an operation acting on a physical device. Such an operation is handled by using one of the (overloaded) predefined procedures SEND_CONTROL and RECEIVE_CONTROL.

2   A procedure SEND_CONTROL may be used to send control information to a physical device. A procedure RECEIVE_CONTROL may be used to monitor the execution of an input-output operation by requesting information from the physical device.

3   Such procedures are declared in the standard package LOW_LEVEL_IO
    and have two parameters identifying the device and the data. However, the
    kinds and formats of the control information will depend on the physical
    characteristics of the machine and the device. Hence, the types of the
    parameters are implementation-defined. Overloaded definitions of these
    procedures should be provided for the supported devices.

4   The visible part of the package defining these procedures is outlined as follows:

5
```
package LOW_LEVEL_IO is
    --  declarations of the possible types for DEVICE and DATA;
    --  declarations of overloaded procedures for these types:
    procedure SEND_CONTROL    (DEVICE : device_type;
                               DATA   : in out data_type);
    procedure RECEIVE_CONTROL (DEVICE : device_type;
                               DATA   : in out data_type);
end;
    32
```

6   The bodies of the procedures SEND_CONTROL and RECEIVE_CONTROL for
    various devices can be supplied in the body of the package LOW_LEVEL_IO.
    These procedure bodies may be written with code statements.

14.7 Example of Input-Output

1   The following example shows the use of some of the text input-output facilities
    in a dialogue with a user at a terminal. The user is prompted to type a color,
    and the program responds by giving the number of items of that color available
    in stock, according to an inventory. The default input and output files are
    used. For simplicity, all the requisite instantiations are given within one
    subprogram; in practice, a package, separate from the procedure, would be
    used.

2
```
with TEXT_IO; use TEXT_IO;
procedure DIALOGUE is
    type COLOR is (WHITE, RED, ORANGE, YELLOW, GREEN, BLUE, BROWN);
    package COLOR_IO is new ENUMERATION_IO(ENUM => COLOR);
    package NUMBER_IO is new INTEGER_IO(INTEGER);
    use COLOR_IO, NUMBER_IO;

    INVENTORY : array (COLOR) of INTEGER := (20, 17, 43, 10,
                                                 28, 173, 87);
    CHOICE : COLOR;
```

---

32 See also Appendix G, AI-00355.

```
procedure ENTER_COLOR (SELECTION : out COLOR) is
begin
   loop
      begin
         PUT("Color selected: ");  --  prompts user
         GET(SELECTION);            --  accepts color typed,
                                    --  or raises exception
         return;
      exception
         when DATA_ERROR =>
            PUT("Invalid color, try again.  ");
            --  user has typed new line
            NEW_LINE(2);
            --  completes execution of the block statement
      end;
   end loop;  --  repeats the block statement until color accepted
end;
begin --  statements of DIALOGUE;

   NUMBER_IO.DEFAULT_WIDTH := 5;

   loop

      ENTER_COLOR(CHOICE);  --  user types color and new line

      SET_COL(5);  PUT(CHOICE); PUT(" items available:");
      SET_COL(40); PUT(INVENTORY(CHOICE));  -- default width is 5
      NEW_LINE;
   end loop;
end DIALOGUE;
```

3 **Example of an interaction (characters typed by the user are italicized):**

```
Color selected: Black
Invalid color, try again.

Color selected: Blue
   BLUE items available:        173
Color selected: Yellow
   YELLOW items available:       10
```

14.7a Example of Additional DEC Ada Input-Output

This example shows the use of the additional DEC Ada input-output package
SEQUENTIAL_MIXED_IO. This program reads values of mixed types from
different file elements and then processes the values for output to an array:

```ada
with SEQUENTIAL_MIXED_IO; use SEQUENTIAL_MIXED_IO;
procedure READ_FILE(FILE_NAME : STRING) is

    type FLOAT_ARRAY is array (INTEGER range <>) of FLOAT;

    F_ARRAY    : FLOAT_ARRAY(1 .. 100);
    F2, F3, F4 : FLOAT;
    I1, I5, I6 : INTEGER;
    LAST       : INTEGER;
    FILE       : FILE_TYPE;

    -- Instantiate GET_ITEM for each type in the file.
    --
    procedure GET_INTEGER is new GET_ITEM(INTEGER);
    procedure GET_FLOAT is new GET_ITEM(FLOAT);
    procedure GET_STRING is new GET_ITEM(STRING);

    -- Instantiate GET_ARRAY for a sequence of floating values.
    --
    procedure GET_FLOAT_ARRAY is
       new GET_ARRAY(FLOAT, INTEGER, FLOAT_ARRAY);

    -- Procedure for processing the values
    --
    procedure PROCESS_ARRAY(FA   : FLOAT_ARRAY;
                            LAST : INTEGER) is separate;

begin
    OPEN(FILE, IN_FILE, FILE_NAME);

    -- Read the first file element consisting of an integer and
    -- two floating point values.
    --
    READ(FILE);
    GET_INTEGER(FILE, I1);
    GET_FLOAT(FILE, F2);
    GET_FLOAT(FILE, F3);

    -- Read the second element consisting of a floating point value
    -- and two integers.
    --
    READ(FILE);
    GET_FLOAT(FILE, F4);
    GET_INTEGER(FILE, I5);
    GET_INTEGER(FILE, I6);
```

```ada
      -- The remainder of the file consists of a number of elements
      -- (each of which contains a variable number of F_floating
      -- values), which are processed up to 100 at a time.
      --
      READ(FILE);
      loop
         if END_OF_BUFFER(FILE) then
            READ(FILE);
         end if;
         GET_FLOAT_ARRAY(FILE, F_ARRAY, LAST);
         PROCESS_ARRAY(F_ARRAY, LAST);
      end loop;
      CLOSE(FILE);

exception
   when END_ERROR => CLOSE(FILE);
end READ_FILE;
```

A

Predefined Language Attributes

1   This annex summarizes the definitions given elsewhere of the predefined language attributes.

2   P′ADDRESS   For a prefix P that denotes an object, a program unit, a label, or an entry:

Yields the address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, this value refers to the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, the value refers to the corresponding hardware interrupt. The value of this attribute is of the type ADDRESS defined in the package SYSTEM. (See 13.7.2.)

3   P′AFT   For a prefix P that denotes a fixed point subtype:

Yields the number of decimal digits needed after the point to accommodate the precision of the subtype P, unless the delta of the subtype P is greater than 0.1, in which case the attribute yields the value one. (P′AFT is the smallest positive integer N for which (10**N)*P′DELTA is greater than or equal to one.) The value of this attribute is of the type *universal_integer.* (See 3.5.10.)

	P'AST_ENTRY	On OpenVMS systems only.
		For a prefix P that denotes a single entry of an object of a task type or an entry of a single task:
		Yields a value of the type AST_HANDLER (in the package SYSTEM) that transforms an AST occurrence into a call of the given entry. This attribute can only occur in a compilation unit to which the package SYSTEM applies, and the pragma AST_ENTRY must have been specified for the entry. (See 9.12a.)
4	P'BASE	For a prefix P that denotes a type or subtype:
		This attribute denotes the base type of P. It is only allowed as the prefix of the name of another attribute: for example, P'BASE'FIRST. (See 3.3.3.)
	P'BIT	For a prefix P that denotes an object:
		Yields the bit offset within the storage unit (byte) containing the first bit of the storage for the object P. The value of this attribute is of the type *universal_integer*. (See 13.7.2.)
5	P'CALLABLE	For a prefix P that is appropriate for a task type:
		Yields the value FALSE when the execution of the task P is either completed or terminated, or when the task is abnormal; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 9.9.)
6	P'CONSTRAINED	For a prefix P that denotes an object of a type with discriminants:
		Yields the value TRUE if a discriminant constraint applies to the object P, or if the object is a constant (including a formal parameter or generic formal parameter of mode **in**); yields the value FALSE otherwise.

If P is a generic formal parameter of mode **in out**, or if P is a formal parameter of mode **in out** or **out** and the type mark given in the corresponding parameter specification denotes an unconstrained type with discriminants, then the value of this attribute is obtained from that of the corresponding actual parameter. The value of this attribute is of the predefined type BOOLEAN. (See 3.7.4.)

7   P′CONSTRAINED   For a prefix P that denotes a private type or subtype:

Yields the value FALSE if P denotes an unconstrained nonformal private type with discriminants; also yields the value FALSE if P denotes a generic formal private type and the associated actual subtype is either an unconstrained type with discriminants or an unconstrained array type; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 7.4.2.)

8   P′COUNT   For a prefix P that denotes an entry of a task unit:

Yields the number of entry calls presently queued on the entry (if the attribute is evaluated within an accept statement for the entry P, the count does not include the calling task). The value of this attribute is of the type *universal_integer.* (See 9.9.)

9   P′DELTA   For a prefix P that denotes a fixed point subtype:

Yields the value of the delta specified in the fixed accuracy definition for the subtype P. The value of this attribute is of the type *universal_real.* (See 3.5.10.)

10	P'DIGITS	For a prefix P that denotes a floating point subtype:
		Yields the number of decimal digits in the decimal mantissa of model numbers of the subtype P. (This attribute yields the number D of section 3.5.7.) The value of this attribute is of the type *universal_integer.* (See 3.5.8.)
11	P'EMAX	For a prefix P that denotes a floating point subtype:
		Yields the largest exponent value in the binary canonical form of model numbers of the subtype P. (This attribute yields the product 4*B of section 3.5.7.) The value of this attribute is of the type *universal_integer.* (See 3.5.8.)
12	P'EPSILON	For a prefix P that denotes a floating point subtype:
		Yields the absolute value of the difference between the model number 1.0 and the next model number above, for the subtype P. The value of this attribute is of the type *universal_real.* (See 3.5.8.)
13	P'FIRST	For a prefix P that denotes a scalar type, or a subtype of a scalar type:
		Yields the lower bound of P. The value of this attribute has the same type as P. (See 3.5.)
14	P'FIRST	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the lower bound of the first index range. The value of this attribute has the same type as this lower bound. (See 3.6.2 and 3.8.2.)
15	P'FIRST(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:

Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound. The argument N must be a static expression of type *universal_integer.* The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

16   P′FIRST_BIT   For a prefix P that denotes a component of a record object:

Yields the offset, from the start of the first of the storage units occupied by the component, of the first bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type *universal_integer.* (See 13.7.2.)

17   P′FORE   For a prefix P that denotes a fixed point subtype:

Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype P, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least two.) The value of this attribute is of the type *universal_integer.* (See 3.5.10.)

18   P′IMAGE   For a prefix P that denotes a discrete type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the predefined type STRING. The result is the *image* of the value of X, that is, a sequence of characters representing the value in display form. The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent, or trailing spaces; but with a one

character prefix that is either a minus sign or a space.

The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. The image of a character other than a graphic character is implementation-defined. (See 3.5.5.)

19	P'LARGE	For a prefix P that denotes a real subtype:
		The attribute yields the largest positive model number of the subtype P. The value of this attribute is of the type *universal_real.* (See 3.5.8 and 3.5.10.)
20	P'LAST	For a prefix P that denotes a scalar type, or a subtype of a scalar type:
		Yields the upper bound of P. The value of this attribute has the same type as P. (See 3.5.)
21	P'LAST	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound. (See 3.6.2 and 3.8.2.)
22	P'LAST(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound. The argument N must be a static expression of type *universal_integer.* The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)

23	P'LAST_BIT	For a prefix P that denotes a component of a record object:
		Yields the offset, from the start of the first of the storage units occupied by the component, of the last bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type *universal_integer.* (See 13.7.2.)
24	P'LENGTH	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the number of values of the first index range (zero for a null range). The value of this attribute is of the type *universal_integer.* (See 3.6.2.)
25	P'LENGTH(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the number of values of the N-th index range (zero for a null range). The value of this attribute is of the type *universal_integer.* The argument N must be a static expression of type *universal_integer.* The value of N must be positive (nonzero) and no greater than the dimensionality of the array. (See 3.6.2 and 3.8.2.)
26	P'MACHINE_EMAX	For a prefix P that denotes a floating point type or subtype:
		Yields the largest value of *exponent* for the machine representation of the base type of P. The value of this attribute is of the type *universal_integer.* (See 13.7.3.)
27	P'MACHINE_EMIN	For a prefix P that denotes a floating point type or subtype:
		Yields the smallest (most negative) value of *exponent* for the machine representation of the base type of P. The value of this

attribute is of the type *universal_integer.* (See 13.7.3.)

28	P′MACHINE_MANTISSA	For a prefix P that denotes a floating point type or subtype:

Yields the number of digits in the *mantissa* for the machine representation of the base type of P (the digits are extended digits in the range 0 to P′MACHINE_RADIX − 1). The value of this attribute is of the type *universal_integer.* (See 13.7.3.)

29	P′MACHINE_OVERFLOWS	For a prefix P that denotes a real type or subtype:

Yields the value TRUE if every predefined operation on values of the base type of P either provides a correct result, or raises the exception NUMERIC_ERROR in overflow situations; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 13.7.3.)

30	P′MACHINE_RADIX	For a prefix P that denotes a floating point type or subtype:

Yields the value of the *radix* used by the machine representation of the base type of P. The value of this attribute is of the type *universal_integer.* (See 13.7.3.)

31	P′MACHINE_ROUNDS	For a prefix P that denotes a real type or subtype:

Yields the value TRUE if every predefined arithmetic operation on values of the base type of P either returns an exact result or performs rounding; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 13.7.3.)

	P'MACHINE_SIZE	For a prefix P that denotes any type or subtype:
		Yields the number of machine bits to be allocated for variables of the type or subtype. This value takes into account any padding bits used by DEC Ada when allocating a variable on a byte boundary. The value of this attribute is of the type *universal_integer*. (See 13.7.2.)
32	P'MANTISSA	For a prefix P that denotes a real subtype:
		Yields the number of binary digits in the binary mantissa of model numbers of the subtype P. (This attribute yields the number B of section 3.5.7 for a floating point type, or of section 3.5.9 for a fixed point type.) The value of this attribute is of the type *universal_integer.* (See 3.5.8 and 3.5.10.)
	P'NULL_PARAMETER	For a prefix P that denotes any type or subtype:
		Yields an (imaginary) object of type or subtype P allocated at (machine) address zero. The attribute is allowed only as the default expression of a formal parameter or as an actual expression of a subprogram call; in either case, the subprogram must be imported. (See 13.9a.1.2.)
33	P'POS	For a prefix P that denotes a discrete type or subtype:
		This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type *universal_integer.* The result is the position number of the value of the actual parameter. (See 3.5.5.)

34	P′POSITION	For a prefix P that denotes a component of a record object:[1]
		Yields the offset, from the start of the first storage unit occupied by the record, of the first of the storage units occupied by the component. This offset is measured in storage units. The value of this attribute is of the type *universal_integer*. (See 13.7.2.)
35	P′PRED	For a prefix P that denotes a discrete type or subtype:
		This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one less than that of X. The exception CONSTRAINT_ERROR is raised if X equals P′BASE′FIRST. (See 3.5.5.)
36	P′RANGE	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the first index range of P, that is, the range P′FIRST .. P′LAST. (See 3.6.2.)
37	P′RANGE(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:
		Yields the N-th index range of P, that is, the range P′FIRST(N) .. P′LAST(N). (See 3.6.2.)
38	P′SAFE_EMAX	For a prefix P that denotes a floating point type or subtype:
		Yields the largest exponent value in the binary canonical form of safe numbers of the base type of P. (This attribute yields the number E of section 3.5.7.) The value of this attribute is of the type *universal_integer*. (See 3.5.8.)

---

[1] See also Appendix G, AI-00258.

39	P'SAFE_LARGE	For a prefix P that denotes a real type or subtype:
		Yields the largest positive safe number of the base type of P. The value of this attribute is of the type *universal_real.* (See 3.5.8 and 3.5.10.)
40	P'SAFE_SMALL	For a prefix P that denotes a real type or subtype:
		Yields the smallest positive (nonzero) safe number of the base type of P. The value of this attribute is of the type *universal_real.* (See 3.5.8 and 3.5.10.)
41	P'SIZE	For a prefix P that denotes an object:
		Yields the number of bits allocated to hold the object. The value of this attribute is of the type *universal_integer.* (See 13.7.2.)
42	P'SIZE	For a prefix P that denotes any type or subtype:
		Yields the minimum number of bits that is needed by the implementation to hold any possible object of the type or subtype P. The value of this attribute is of the type *universal_integer.* (See 13.7.2.)
43	P'SMALL	For a prefix P that denotes a real subtype:
		Yields the smallest positive (nonzero) model number of the subtype P. The value of this attribute is of the type *universal_real.* (See 3.5.8 and 3.5.10.)

44	P′STORAGE_SIZE	For a prefix P that denotes an access type or subtype:
		Yields the total number of storage units reserved for the collection associated with the base type of P. The value of this attribute is of the type *universal_integer.* (See 13.7.2.)
45	P′STORAGE_SIZE	For a prefix P that denotes a task type or a task object:
		Yields the number of storage units reserved for each activation of a task of the type P or for the activation of the task object P. The value of this attribute is of the type *universal_integer.* (See 13.7.2.)
46	P′SUCC	For a prefix P that denotes a discrete type or subtype:
		This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one greater than that of X. The exception CONSTRAINT_ERROR is raised if X equals P′BASE′LAST. (See 3.5.5.)
47	P′TERMINATED	For a prefix P that is appropriate for a task type:
		Yields the value TRUE if the task P is terminated; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN. (See 9.9.)
	P′TYPE_CLASS	For a prefix P that denotes a type or subtype:
		Yields the value of the type class for the full type of P. If P is a generic formal type, then the value is that for the corresponding actual subtype. The value of this attribute is of the type TYPE_CLASS in the package SYSTEM. (See 13.7a.2.)

48	P´VAL	For a prefix P that denotes a discrete type or subtype:
		This attribute is a special function with a single parameter X which can be of any integer type. The result type is the base type of P. The result is the value whose position number is the *universal_integer* value corresponding to X. The exception CONSTRAINT_ERROR is raised if the *universal_integer* value corresponding to X is not in the range P´POS(P´BASE´FIRST) .. P´POS(P´BASE´LAST). (See 3.5.5.)
49	P´VALUE	For a prefix P that denotes a discrete type or subtype:
		This attribute is a function with a single parameter. The actual parameter X must be a value of the predefined type STRING. The result type is the base type of P. Any leading and any trailing spaces of the sequence of characters that corresponds to X are ignored.
		For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of P, the result is the corresponding enumeration value. For an integer type, if the sequence of characters has the syntax of an integer literal, with an optional single leading character that is a plus or minus sign, and if there is a corresponding value in the base type of P, the result is this value. In any other case, the exception CONSTRAINT_ERROR is raised. (See 3.5.5.)
50	P´WIDTH	For a prefix P that denotes a discrete subtype:

Yields the maximum image length over all values of the subtype P (the *image* is the sequence of characters returned by the attribute IMAGE). The value of this attribute is of the type *universal_integer*. (See 3.5.5.)

B

Predefined Language Pragmas

1   This annex defines the pragmas LIST, PAGE, and OPTIMIZE, and summarizes the definitions given elsewhere of the remaining language-defined pragmas.

The DEC Ada pragmas IDENT and TITLE are also defined in this annex.

Pragma	Meaning
AST_ENTRY	On OpenVMS systems only.
	Takes the simple name of a single entry as the single argument; at most one AST_ENTRY pragma is allowed for any given entry. This pragma must be used in combination with the AST_ENTRY attribute, and is only allowed after the entry declaration and in the same task type specification or single task as the entry to which it applies. This pragma specifies that the given entry may be used to handle an OpenVMS asynchronous system trap (AST) resulting from an OpenVMS system service call. The pragma does not affect normal use of the entry (see 9.12a).
COMMON_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a linker storage area) and a size as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a variable declared by an earlier declarative

item of the same declarative part or package specification. The variable must have a size that is known at compile time, and it must not require implicit initialization. This pragma is not allowed for objects declared with a renaming declaration. This pragma enables the shared use of objects that are stored in overlaid storage areas (see 13.9a.2.3).

COMPONENT_ALIGNMENT	Takes an alignment choice and optionally the simple name of an array or record type as arguments. When no simple name is specified, the pragma must occur within a declarative part or package specification, and the effect of the pragma extends to types declared from the place of the pragma to the end of the innermost declarative part or package specification in which the pragma was declared. When a simple name is specified, the pragma and the type declaration must both occur immediately within the same declarative part, package specification, or task specification; the declaration must occur before the pragma. The position of the pragma and the restrictions on the named type are governed by the same rules as those for a representation clause. This pragma specifies the kind of alignment used for the components of the array or record types to which it applies (see 13.1a).

CONTROLLED

Takes the simple name of an access type as the single argument. This pragma is only allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. This pragma is not allowed for

a derived type. This pragma specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (see 4.8).

3   ELABORATE

Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause. This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit (see 10.5).

EXPORT_EXCEPTION

On OpenVMS systems only.

Takes an internal name denoting an exception, and optionally takes an external designator (the name of a linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a condition code) as arguments. A code value must be specified when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part

or package specification; it is not allowed for an exception declared with a renaming declaration or for an exception declared in a generic unit. This pragma permits an Ada exception to be handled by programs written in another programming language (see 13.9a.3.2).

EXPORT_FUNCTION	Takes an internal name denoting a function, and optionally takes an external designator (the name of a linker global symbol), parameter types, result type, parameter mechanisms, and result mechanism as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is not allowed for a function declared with a renaming declaration, and it is not allowed for a generic function (it may be given for a generic instantiation). This pragma permits an Ada function to be called from a program written in another programming language (see 13.9a.1.3).
EXPORT_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of a linker global symbol) and size option (a linker absolute global symbol that will be defined in the object module—useful on OpenVMS systems only) as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a constant or a variable

	declared by an earlier declarative item of the same declarative part or package specification; the declaration must occur at the outermost level of a library package specification or body. The object to be exported must have a size that is known at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits an Ada object to be referred to by a routine written in another programming language (see 13.9a.2.2).
EXPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is not allowed for a procedure declared with a renaming declaration, and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada routine to be called from a program written in another programming language (see 13.9a.1.3).
EXPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms

as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode **out**. This pragma is not allowed for a procedure declared with a renaming declaration and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada procedure to behave as a function that both returns a value and causes side effects on its parameters when it is called from a routine written in another programming language (see 13.9a.1.3).

FLOAT_REPRESENTATION

On OpenVMS and Digital UNIX systems only.

On OpenVMS VAX systems, takes VAX_FLOAT as the single argument. On OpenVMS Alpha systems, takes either VAX_FLOAT or IEEE_FLOAT as the single argument; the default is VAX_FLOAT. On Digital UNIX systems, takes IEEE_FLOAT as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. It specifies the choice of representation to be used for the predefined floating point types in the package STANDARD (see 3.5.7a).

IDENT	Takes a string literal of 31 or fewer characters as the single argument. The pragma IDENT has the following form:

```
pragma IDENT (string_literal);
```

This pragma is allowed only in the outermost declarative part or declarative items of a compilation unit. The given string is used to identify the object module associated with the compilation unit in which the pragma IDENT occurs.

IMPORT_EXCEPTION	On OpenVMS systems only.

Takes an internal name denoting an exception, and optionally takes an external designator (the name of a linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a condition code) as arguments. A code value is allowed only when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not allowed for an exception declared with a renaming declaration. This pragma permits a non-Ada exception (most notably, an OpenVMS condition) to be handled by an Ada program (see 13.9a.3.1).

IMPORT_FUNCTION	Takes an internal name denoting a function, and optionally takes an external designator (the name of a linker global symbol), parameter types, result type, parameter mechanisms, and result mechanism as arguments. On OpenVMS systems, a first optional parameter is also available as an

argument. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is allowed for a function declared with a renaming declaration; it is not allowed for a generic function or a generic function instantiation. This pragma permits a non-Ada routine to be used as an Ada function (see 13.9a.1.1).

IMPORT_OBJECT   Takes an internal name denoting an object, and optionally takes an external designator (the name of a linker global symbol) and size (a linker absolute global symbol that will be defined in the object module—useful on OpenVMS systems only) as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a variable declared by an earlier declarative item of the same declarative part or package specification. The variable must have a size that is known at compile time, and it cannot have an initial value. This pragma is not allowed for objects declared with a renaming declaration. This pragma permits storage declared in a non-Ada routine to be referred to by an Ada program (see 13.9a.2.1).

IMPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. On OpenVMS systems, a first optional parameter is also available as an argument. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure or a generic procedure instantiation. This pragma permits a non-Ada routine to be used as an Ada procedure (see 13.9a.1.1).
IMPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a linker global symbol), parameter types, and parameter mechanisms as arguments. On OpenVMS systems, a first optional parameter is also available as an argument. The pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the

pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode **out**. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure. This pragma permits a non-Ada routine that returns a value and causes side effects on its parameters to be used as an Ada procedure (see 13.9a.1.1).

4 INLINE

Takes one or more names as arguments; each name is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations (see 6.3.2).

INLINE_GENERIC

Takes one or more names as arguments; each name is either the name of a generic declaration or the name of an instance of a generic declaration. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. Each argument must be the simple name of a generic subprogram or package, or a (nongeneric) subprogram or package that is an instance of a generic subprogram or package declared by

an earlier declarative item of the same declarative part or package specification. This pragma specifies that inline expansion of the generic body is desired for each instantiation of the named generic declarations or of the particular named instances; the pragma does not apply to calls of instances of generic subprograms (see 12.1a).

5 INTERFACE

Takes a language name and a subprogram name as arguments. This pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. This pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram (see 13.9).

In DEC Ada, the pragma INTERFACE is required in combination with the pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, IMPORT_ VALUED_PROCEDURE, and INTERFACE_NAME when any of those pragmas are used (see 13.9a.1).

INTERFACE_NAME

Takes an internal name and an external name as arguments. The internal name may be an Ada simple name that denotes a subprogram or an object. If the declared entity is a function, the internal name may be a string literal that denotes an operator

symbol. The external name may be any string literal; the literal is used as a linker global symbol that is associated with the external subprogram or object. This pragma is only allowed at the place of a declarative item, and must apply to an entity declared by an earlier declarative item of the same declarative part or package specification.

If this pragma applies to a subprogram, then the pragma INTERFACE must also apply (see 13.9). If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit. This pragma is allowed for subprograms declared with a renaming declaration. This pragma is not allowed for a generic subprogram or a generic subprogram instantiation.

If this pragma applies to an object, then the size of the object must be known at compile time. This pragma is not allowed for an object declared with a renaming declaration.

This pragma associates an external symbol with the internal Ada name for a subprogram or object (see 13.9b).

6   LIST

Takes one of the identifiers ON or OFF as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

LONG_FLOAT	On OpenVMS systems only. Also, the value of the pragma FLOAT_ REPRESENTATION must be VAX_ FLOAT.
	Takes either D_FLOAT or G_FLOAT as the single argument. The default is G_FLOAT. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. It specifies the choice of representation to be used for the predefined type LONG_FLOAT in the package STANDARD, and for floating point type declarations with **digits** specified in the range 7 .. 15 (see 3.5.7b).
MAIN_STORAGE	On OpenVMS VAX systems only.
	Takes one or two nonnegative static simple expressions of some integer type as arguments. This pragma is only allowed in the outermost declarative part of a library subprogram; at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma causes a fixed-size stack to be created for a main task (the task associated with a main program), and determines the number of storage units (bytes) to be allocated for the stack working storage area or guard pages or both. The value specified for either or both the working storage area and guard pages is rounded up to an integral number of pages. A value of zero for the working storage area results in the use of a default size; a value of zero for the guard pages results in no guard storage. A negative value for either

		working storage or guard pages causes the pragma to be ignored (see 13.2b).
7	MEMORY_SIZE	Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE (see 13.7).
8	OPTIMIZE	Takes one of the identifiers TIME or SPACE as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion.

In DEC Ada, this pragma is only allowed immediately within a declarative part of a body declaration. |
9	PACK	Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type (see 13.1).
10	PAGE	This pragma has no argument, and is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).
	PASSIVE	On OpenVMS Alpha and Digital UNIX systems only.

Takes an argument indicating whether or not to make a containing task passive. The default (no arguments) indicates a desire to make the containing task passive. This pragma must be specified within a task specification. The specification can be for a task type or for a single task. (see 9.8b).

11   PRIORITY   Takes a static expression of the predefined integer subtype PRIORITY as the single argument. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program. It specifies the priority of the task (or tasks of the task type) or the priority of the main program (see 9.8).

PSECT_OBJECT   On OpenVMS systems only.

Has the same syntax and the same effect as the pragma COMMON_OBJECT (see 13.9a.2.3).

12   SHARED   Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. This pragma specifies that every read or update of the variable is a synchronization point for that variable. An implementation must restrict the objects for which this pragma is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation (see 9.11).

SHARE_GENERIC

Takes one or more names as arguments; each name is either the name of a generic declaration or the name of an instance of a generic declaration. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. Each argument either must be the simple name of a generic subprogram or package, or it must be a (nongeneric) subprogram or package that is an instance of a generic subprogram or package. If the argument is an instance of a generic subprogram or package, then it must be declared by an earlier declarative item of the same declarative part or package specification. This pragma specifies that generic code sharing is desired for each instantiation of the named generic declarations or of the particular named instances (see 12.1b).

13 STORAGE_UNIT

Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number STORAGE_UNIT (see 13.7).

In DEC Ada, the only argument allowed for this pragma is 8 (bits).

14 SUPPRESS

Takes as arguments the identifier of a check and optionally also the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a

declarative part or immediately within
a package specification. In the latter
case, the only allowed form is with a
name that denotes an entity (or several
overloaded subprograms) declared
immediately within the package
specification. The permission to omit
the given check extends from the
place of the pragma to the end of the
declarative region associated with the
innermost enclosing block statement or
program unit. For a pragma given in a
package specification, the permission
extends to the end of the scope of the
named entity.

If the pragma includes a name, the
permission to omit the given check
is further restricted: it is given only
for operations on the named object
or on all objects of the base type of a
named type or subtype; for calls of a
named subprogram; for activations of
tasks of the named task type; or for
instantiations of the given generic unit
(see 11.7).

SUPPRESS_ALL   This pragma has no argument and is
                      only allowed following a compilation
                      unit. This pragma specifies that
                      all run-time checks in the unit are
                      suppressed (see 11.7).

15   SYSTEM_NAME   Takes an enumeration literal as the
                      single argument. This pragma is only
                      allowed at the start of a compilation,
                      before the first compilation unit
                      (if any) of the compilation. The effect of
                      this pragma is to use the enumeration
                      literal with the specified identifier
                      for the definition of the constant
                      SYSTEM_NAME. This pragma is
                      only allowed if the specified identifier
                      corresponds to one of the literals of the

	type NAME declared in the package SYSTEM (see 13.7).
TASK_STORAGE	Takes the simple name of a task type and a static expression of some integer type as arguments. This pragma is allowed anywhere that a task storage specification is allowed; that is, the declaration of the task type to which the pragma applies and the pragma must both occur (in this order) immediately within the same declarative part, package specification, or task specification. The effect of this pragma is to use the value of the expression as the number of storage units (bytes) to be allocated as guard storage. The value is rounded up to an appropriate boundary. A negative value causes the pragma to be ignored. A zero value has system-specific results: on OpenVMS VAX systems, a value of zero results in no guard storage; on OpenVMS Alpha and DEC OSF/1 or ULTRIX systems, a value of zero results in a minimal guard area (see 13.2a).
TIME_SLICE	On OpenVMS and Digital UNIX systems only.
	Takes a static expression of the predefined fixed point type DURATION (in the package STANDARD) as the single argument. This pragma is only allowed in the outermost declarative part of a library subprogram, and at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma causes the task scheduler to turn time slicing on or off and, on some systems, to limit the

amount of continuous execution time given to a task (see 9.8a; see also the appropriate run-time reference manual for implementation differences across systems).

TITLE

Takes a title or a subtitle string, or both, as arguments. The pragma TITLE has the following form:

```
pragma TITLE (titling-option
    [,titling-option]);

titling-option :=
    [TITLE =>] string_literal
  | [SUBTITLE =>] string_literal
```

This pragma is allowed anywhere a pragma is allowed; the given strings supersede the default title and/or subtitle portions of a compilation listing.

VOLATILE

Takes the simple name of a variable as the single argument. This pragma is only allowed for a variable declared by an object declaration. The variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. The pragma must appear before any occurrence of the name of the variable other than in an address clause or in one of the DEC Ada pragmas IMPORT_OBJECT, EXPORT_OBJECT, COMMON_OBJECT, or PSECT_OBJECT. The variable cannot be declared by a renaming declaration. The pragma VOLATILE specifies that the variable may be modified asynchronously. This pragma instructs the compiler to obtain the value of a variable from memory each time it is used (see 9.11).

C

Predefined Language Environment

1 This annex outlines the specification of the package STANDARD containing all predefined identifiers in the language. The corresponding package body is implementation-defined and is not shown.

2 The operators that are predefined for the types declared in the package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal_real*) and for undefined information (such as *implementation_defined* and *any_fixed_point_ type*).

3   **package** STANDARD **is**

4   **type** BOOLEAN **is** (FALSE, TRUE);

```
-- The predefined relational operators for this type are
-- as follows:

-- function "="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

-- The predefined logical operators and the predefined logical
-- negation operator are as follows:

-- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

-- function "not" (RIGHT : BOOLEAN) return BOOLEAN;
```

5   `-- The universal type` *universal_integer* `is predefined.`

```
          type LONG_INTEGER is implementation_defined;

6         type INTEGER is implementation_defined;

          -- The predefined operators for this type are as follows:

          -- function "="   (LEFT, RIGHT : INTEGER) return BOOLEAN;
          -- function "/="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
          -- function "<"   (LEFT, RIGHT : INTEGER) return BOOLEAN;
          -- function "<="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
          -- function ">"   (LEFT, RIGHT : INTEGER) return BOOLEAN;
          -- function ">="  (LEFT, RIGHT : INTEGER) return BOOLEAN;

          -- function "+"   (RIGHT : INTEGER) return INTEGER;
          -- function "-"   (RIGHT : INTEGER) return INTEGER;
          -- function "abs" (RIGHT : INTEGER) return INTEGER;

          -- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
          -- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
          -- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
          -- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
          -- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
          -- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

          -- function "**"  (LEFT  : INTEGER;
          --                 RIGHT : INTEGER) return INTEGER;

7     -- An implementation may provide additional predefined integer types.
      -- It is recommended that the names of such additional types end
      -- with INTEGER as in SHORT_INTEGER  or  LONG_INTEGER. The specifi-
      -- cation of each operator for the type universal_integer, or for
      -- any additional predefined integer type, is obtained by replacing
      -- INTEGER by the name of the type  in  the specification of the
      -- corresponding  operator  of the type INTEGER, except for the
      -- right operand of the exponentiating operator.

          type SHORT_INTEGER is implementation_defined;
          type SHORT_SHORT_INTEGER is implementation_defined;

8     -- The universal type universal_real is predefined.

9         type FLOAT is implementation_defined;

          -- The predefined operators for this type are as follows:

          -- function "="   (LEFT, RIGHT : FLOAT) return BOOLEAN;
          -- function "/="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
          -- function "<"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
          -- function "<="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
          -- function ">"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
          -- function ">="  (LEFT, RIGHT : FLOAT) return BOOLEAN;

          -- function "+"   (RIGHT : FLOAT) return FLOAT;
          -- function "-"   (RIGHT : FLOAT) return FLOAT;
          -- function "abs" (RIGHT : FLOAT) return FLOAT;
```

```
              -- function "+"   (LEFT, RIGHT : FLOAT) return FLOAT;
              -- function "-"   (LEFT, RIGHT : FLOAT) return FLOAT;
              -- function "*"   (LEFT, RIGHT : FLOAT) return FLOAT;
              -- function "/"   (LEFT, RIGHT : FLOAT) return FLOAT;

              -- function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;

    10    -- An implementation may provide additional predefined floating point
          -- types. It is recommended that the names of such additional types
          -- end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.  The specification
          -- of each operator for the type universal_real, or for any additional
          -- predefined  floating point type, is obtained by replacing FLOAT by
          -- the name of the type in the specification of the corresponding
          -- operator of the type FLOAT.

              type LONG_FLOAT is implementation_defined;
              type LONG_LONG_FLOAT is implementation_defined;

    11    -- In addition, the following operators are predefined for
          -- universal types:

              -- function "*" (LEFT  : universal_integer;
              --               RIGHT : universal_real)    return universal_real;
              -- function "*" (LEFT  : universal_real;
              --               RIGHT : universal_integer) return universal_real;
              -- function "/" (LEFT  : universal_real;

           type {universal_fixed} is {delta unbounded range unbounded .. unbounded};

              -- The type universal_fixed is predefined.  The only operators
              -- declared for this type are

              -- function "*" (LEFT  : any_fixed_point_type;
              --               RIGHT : any_fixed_point_type)
              --    return universal_fixed;
              -- function "/" (LEFT  : any_fixed_point_type;
              --               RIGHT : any_fixed_point_type)
              --    return universal_fixed;

    12    -- The following characters form the standard Latin-1 character set.
          -- Character  literals corresponding to control characters are not
          -- identifiers;  they are indicated in italics in this definition.
```

```
13    type CHARACTER is

     (nul,  soh,  stx,  etx,    eot,  enq,  ack,  bel,
      bs,   ht,   lf,   vt,     ff,   cr,   so,   si,
      dle,  dc1,  dc2,  dc3,    dc4,  nak,  syn,  etb,
      can,  em,   sub,  esc,    fs,   gs,   rs,   us,
      ' ',  '!',  '"',  '#',    '$',  '%',  '&',  ''',
      '(',  ')',  '*',  '+',    ',',  '-',  '.',  '/',
      '0',  '1',  '2',  '3',    '4',  '5',  '6',  '7',
      '8',  '9',  ':',  ';',    '<',  '=',  '>',  '?',

      '@',  'A',  'B',  'C',    'D',  'E',  'F',  'G',
      'H',  'I',  'J',  'K',    'L',  'M',  'N',  'O',
      'P',  'Q',  'R',  'S',    'T',  'U',  'V',  'W',
      'X',  'Y',  'Z',  '[',    '\',  ']',  '^',  '_',

      '`',  'a',  'b',  'c',    'd',  'e',  'f',  'g',
      'h',  'i',  'j',  'k',    'l',  'm',  'n',  'o',
      'p',  'q',  'r',  's',    't',  'u',  'v',  'w',
      'x',  'y',  'z',  '{',    '|',  '}',  '~',  del,

      x80,  x81,  bph,  nbh,    ind,  nel,  ssa,  esa,
      hts,  htj,  vts,  pld,    plu,  ri,   ss2,  ss3,
      dcs,  pu1,  pu2,  sts,    cch,  mw,   spa,  epa,
      sos,  x99,  sci,  csi,    st,   osc,  pm,   apc,

      ' ',  '¡',  '¢',  '£',    '¤',  '¥',  '|',  '§',
      '¨',  '©',  'ª',  '«',    '¬',  '',  '®',  '¯',
      '°',  '±',  '²',  '³',    '´',  'µ',  '¶',  '·',
      '¸',  '¹',  'º',  '»',    '¼',  '½',  '¾',  '¿',

      'À',  'Á',  'Â',  'Ã',    'Ä',  'Å',  'Æ',  'Ç',
      'È',  'É',  'Ê',  'Ë',    'Ì',  'Í',  'Î',  'Ï',
      'Ð',  'Ñ',  'Ò',  'Ó',    'Ô',  'Õ',  'Ö',  '× ',
      'Ø',  'Ù',  'Ú',  'Û',    'Ü',  'Ý',  'Þ',  'ß',

      'à',  'á',  'â',  'ã',    'ä',  'å',  'æ',  'ç',
      'è',  'é',  'ê',  'ë',    'ì',  'í',  'î',  'ï',
      'ð',  'ñ',  'ò',  'ó',    'ô',  'õ',  'ö',  '÷ ',
      'ø',  'ù',  'ú',  'û',    'ü',  'ý',  'þ',  'ÿ');

  -- for CHARACTER use  --  256 Latin-1 character set without holes
  --      (0, 1, 2, 3, 4, 5, ..., 253, 254, 255);

  for CHARACTER'SIZE use 8;
```

14   -- The predefined operators for the type CHARACTER are the same
            -- as for any enumeration type.

15   **package** ASCII **is**

            -- Control characters:

```
NUL       : constant CHARACTER := nul;
SOH       : constant CHARACTER := soh;
STX       : constant CHARACTER := stx;
ETX       : constant CHARACTER := etx;
EOT       : constant CHARACTER := eot;
ENQ       : constant CHARACTER := enq;
ACK       : constant CHARACTER := ack;
BEL       : constant CHARACTER := bel;
BS        : constant CHARACTER := bs;
HT        : constant CHARACTER := ht;
LF        : constant CHARACTER := lf;
VT        : constant CHARACTER := vt;
FF        : constant CHARACTER := ff;
CR        : constant CHARACTER := cr;
SO        : constant CHARACTER := so;
SI        : constant CHARACTER := si;
DLE       : constant CHARACTER := dle;
DC1       : constant CHARACTER := dc1;
DC2       : constant CHARACTER := dc2;
DC3       : constant CHARACTER := dc3;
DC4       : constant CHARACTER := dc4;
NAK       : constant CHARACTER := nak;
SYN       : constant CHARACTER := syn;
ETB       : constant CHARACTER := etb;
CAN       : constant CHARACTER := can;
EM        : constant CHARACTER := em;
SUB       : constant CHARACTER := sub;
ESC       : constant CHARACTER := esc;
FS        : constant CHARACTER := fs;
GS        : constant CHARACTER := gs;
RS        : constant CHARACTER := rs;
US        : constant CHARACTER := us;
DEL       : constant CHARACTER := del;
```

```
            -- Other characters:

            EXCLAM      : constant CHARACTER := '!';
            QUOTATION   : constant CHARACTER := '"';
            SHARP       : constant CHARACTER := '#';
            DOLLAR      : constant CHARACTER := '$';
            PERCENT     : constant CHARACTER := '%';
            AMPERSAND   : constant CHARACTER := '&';
            COLON       : constant CHARACTER := ':';
            SEMICOLON   : constant CHARACTER := ';';
            QUERY       : constant CHARACTER := '?';
            AT_SIGN     : constant CHARACTER := '@';
            L_BRACKET   : constant CHARACTER := '[';
            BACK_SLASH  : constant CHARACTER := '\';
            R_BRACKET   : constant CHARACTER := ']';
            CIRCUMFLEX  : constant CHARACTER := '^';
            UNDERLINE   : constant CHARACTER := '_';
            GRAVE       : constant CHARACTER := '`';
            L_BRACE     : constant CHARACTER := '{';
            BAR         : constant CHARACTER := '|';
            R_BRACE     : constant CHARACTER := '}';
            TILDE       : constant CHARACTER := '~';

            -- Lower case letters:

            LC_A : constant CHARACTER := 'a';
            ...
            LC_Z : constant CHARACTER := 'z';

        end ASCII;
```

16   -- Predefined subtypes:

```
        subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;
        subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

17   -- Predefined string type:

```
        type STRING is array(POSITIVE range <>) of CHARACTER;
        pragma PACK(STRING);
```

18   -- The predefined operators for this type are as follows:

```
        -- function "="  (LEFT, RIGHT : STRING) return BOOLEAN;
        -- function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
        -- function "<"  (LEFT, RIGHT : STRING) return BOOLEAN;
        -- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
        -- function ">"  (LEFT, RIGHT : STRING) return BOOLEAN;
        -- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;
```

```
--    function "&" (LEFT  : STRING;
--                  RIGHT : STRING)    return STRING;
--    function "&" (LEFT  : CHARACTER;
--                  RIGHT : STRING)    return STRING;
--    function "&" (LEFT  : STRING;
--                  RIGHT : CHARACTER) return STRING;
--    function "&" (LEFT  : CHARACTER;
--                  RIGHT : CHARACTER) return STRING;
```

19   **type** DURATION **is delta** *implementation_defined*
            range *implementation_defined*;

        for DURATION'SIZE **use** 32;

        -- The predefined operators for the type DURATION are the same
        -- as for any fixed point type.

20   -- The predefined exceptions:

```
CONSTRAINT_ERROR : exception;
NUMERIC_ERROR    : exception;
PROGRAM_ERROR    : exception;
STORAGE_ERROR    : exception;
TASKING_ERROR    : exception;
```

     end STANDARD;

21   Certain aspects of the predefined entities cannot be completely described in
     the language itself. For example, although the enumeration type BOOLEAN
     can be written showing the two enumeration literals FALSE and TRUE, the
     short-circuit control forms cannot be expressed in the language.

     Note:

22   The language definition predefines the following library units:[1]

     -    The package CALENDAR   (see 9.6)

     -    The package SYSTEM   (see 13.7)
     -    The package MACHINE_CODE (if provided)   (see 13.8)
     -    The generic procedure UNCHECKED_DEALLOCATION   (see 13.10.1)
     -    The generic function UNCHECKED_CONVERSION   (see 13.10.2)

     -    The generic package SEQUENTIAL_IO   (see 14.2.3)

     _____

     [1] See also Appendix G, AI-00355.

- The generic package DIRECT_IO   (see 14.2.5)
- The package TEXT_IO   (see 14.3.10)
- The package IO_EXCEPTIONS   (see 14.5)
- The package LOW_LEVEL_IO   (see 14.6)

D
Glossary

This appendix is informative and is not part of the standard definition of the Ada programming language. Italicized terms in the abbreviated descriptions below either have glossary entries themselves or are described in entries for related terms.

Absolute global symbol (OpenVMS term).

An absolute global symbol is a *global symbol* with a fixed numerical value. The value is not relocatable. It is not changed during linking operations.

Accept statement.

See *entry.*

Access type.

A value of an access type (an *access value*) is either a null value, or a value that *designates* an *object* created by an *allocator.* The designated object can be read and updated via the access value. The definition of an access type specifies the type of the objects designated by values of the access type. See also *collection.*

Actual parameter.

See *parameter.*

Aggregate.

The evaluation of an aggregate yields a value of a *composite type.* The value is specified by giving the value of each of the *components.* Either *positional association* or *named association* may be used to indicate which value is associated with which component.

Allocator.

The evaluation of an allocator creates an *object* and returns a new *access value* which *designates* the object.

Array type.

A value of an array type consists of *components* which are all of the same *subtype* (and hence, of the same type). Each component is uniquely distinguished by an *index* (for a one-dimensional array) or by a sequence of indices (for a multidimensional array). Each index must be a value of a *discrete type* and must lie in the correct index *range.*

Assignment.

Assignment is the *operation* that replaces the current value of a *variable* by a new value. An *assignment statement* specifies a variable on the left, and on the right, an *expression* whose value is to be the new value of the variable.

Asynchronous system trap (OpenVMS term).

An asynchronous system trap (AST) is an interrupt mechanism that is used by some OpenVMS system services to transfer control to a user-specified procedure when an asynchronous event occurs during a process's execution. In DEC Ada, control can be passed to a task *entry* that handles the event.

Attribute.

The evaluation of an attribute yields a predefined characteristic of a named entity; some attributes are *functions.*

Bit array.

A bit array is an *array* whose *components* are not byte aligned, yet which is also not a *bit string.*

Bit string.

A bit string is any one-dimensional *array* of a *discrete type* whose *components* occupy successive single bits.

Block statement.

A block statement is a single statement that may contain a sequence of statements. It may also include a *declarative part*, and *exception handlers*; their effects are local to the block statement.

Body.

A body defines the execution of a *subprogram*, *package*, or *task.* A *body stub* is a form of body that indicates that this execution is defined in a separately compiled *subunit.*

Box.

A box is an Ada delimiter (<>) used to denote an undefined discrete range for *array types* and *generic formal types.*

Collection.

A collection is the entire set of *objects* created by evaluation of *allocators* for an *access type.*

Compilation unit.

A compilation unit is the *declaration* or the *body* of a *program unit*, presented for compilation as an independent text. It is optionally preceded by a *context clause*, naming other compilation units upon which it depends by means of one or more *with clauses.*

Component.

A component is a value that is a part of a larger value, or an *object* that is part of a larger object.

Composite type.

A composite type is one whose values have *components.* There are two kinds of composite type: *array types* and *record types.*

Condition (OpenVMS term).

An OpenVMS exception condition is a hardware- or software-detected event that alters the normal flow of instruction execution.

Condition value (OpenVMS term).

A condition value is a 32-bit value used to uniquely identify an OpenVMS exception *condition.*

Constant.

See *object.*

Constraint.

A constraint determines a subset of the values of a *type.* A value in that subset *satisfies* the constraint.

Context clause.

See *compilation unit.*

Contiguous array.

A contiguous array is one in which the storage of the array *components* is allocated without any separation between adjacent components.

Declaration.

A declaration associates an identifier (or some other notation) with an entity. This association is in effect within a region of text called the *scope* of the declaration. Within the scope of a declaration, there are places where it is possible to use the identifier to refer to the associated declared entity. At such places the identifier is said to be a *simple name* of the entity; the *name* is said to *denote* the associated entity.

Declarative Part.

A declarative part is a sequence of *declarations.* It may also contain related information such as *subprogram bodies* and *representation clauses.*

Denote.

See *declaration.*

Derived Type.

A derived type is a *type* whose operations and values are replicas of those of an existing type. The existing type is called the *parent type* of the derived type.

Descriptor.

A descriptor is a data structure used for parameter passing. The descriptor contains the address, data type, and size of the parameter, as well as other information needed to fully describe the data being passed.

Designate.

See *access type*, *task.*

Direct visibility.

See *visibility.*

Discrete Type.

A discrete type is a *type* which has an ordered set of distinct values. The discrete types are the *enumeration* and *integer types.* Discrete types are used for indexing and iteration, and for choices in case statements and record *variants.*

Discriminant.

A discriminant is a distinguished *component* of an *object* or value of a *record type.* The *subtypes* of other components, or even their presence or absence, may depend on the value of the discriminant.

Discriminant constraint.

A discriminant constraint on a *record type* or *private type* specifies a value for each *discriminant* of the *type.*

Elaboration.

The elaboration of a *declaration* is the process by which the declaration achieves its effect (such as creating an *object*); this process occurs during program execution.

Entry.

An entry is used for communication between *tasks.* Externally, an entry is called just as a *subprogram* is called; its internal behavior is specified by one or more *accept statements* specifying the actions to be performed when the entry is called.

Enumeration type.

An enumeration type is a *discrete type* whose values are represented by enumeration literals which are given explicitly in the *type declaration.* These enumeration literals are either *identifiers* or *character literals.*

Evaluation.

The evaluation of an *expression* is the process by which the value of the expression is computed. This process occurs during program execution.

Exception.

An exception is an error situation which may arise during program execution. To *raise* an exception is to abandon normal program execution so as to signal that the error has taken place. An *exception handler* is a portion of program text specifying a response to the exception. Execution of such a program text is called *handling* the exception.

Expanded name.

An expanded name *denotes* an entity which is *declared* immediately within some construct. An expanded name has the form of a *selected component*: the *prefix* denotes the construct (a *program unit*; or a *block*, loop, or *accept statement*); the *selector* is the *simple name* of the entity.

Expression.

An expression defines the computation of a value.

File descriptor (Digital UNIX term).

A file descriptor is a nonnegative integer that is assigned to each file by the Digital UNIX system. File descriptors can be used to identify files in input-output operations. The file descriptor 0 always refers to Digital UNIX standard input; the file descriptor 1 to standard output, and the file descriptor 2 to standard error.

Fixed point type.

See *real type.*

Floating point type.

See *real type.*

Formal parameter.

See *parameter.*

Function.

See *subprogram.*

Generic unit.

A generic unit is a template either for a set of *subprograms* or for a set of *packages.* A subprogram or package created using the template is called an *instance* of the generic unit. A *generic instantiation* is the kind of *declaration* that creates an instance. A generic unit is written as a subprogram or package but with the specification prefixed by a *generic formal part* which may declare *generic formal parameters.* A generic formal parameter is either a *type*, a *subprogram*, or an *object.* A generic unit is one of the kinds of *program unit.*

Global symbol.

A symbol defined in an OpenVMS module (such as a source, object, or image module) or Digital UNIX object file that is potentially available for reference by another module or object file. The *linker* resolves global symbols. (The linker matches references with definitions).

Handler.

See *exception.*

Index.

See *array type.*

Index constraint.

An index constraint for an *array type* specifies the lower and upper bounds for each index *range* of the array type.

Indexed component.

An indexed component *denotes* a *component* in an *array.* It is a form of *name* containing *expressions* which specify the values of the *indices* of the array component. An indexed component may also denote an *entry* in a family of entries.

Instance.

See *generic unit.*

Instantiation.

An instantiation is the creation of a particular instance of a *generic unit.* In other words, the template defined by the generic *package* or *subprogram* is named, and all generic *parameters* are replaced by actual parameters.

Integer type.

An integer type is a *discrete type* whose values represent all integer numbers within a specific *range.*

Lexical element.

A lexical element is an identifier, a *literal*, a delimiter, or a comment.

Limited type.

A limited type is a *type* for which neither assignment nor the predefined comparison for equality is implicitly declared. All *task* types are limited. A *private type* can be defined to be limited. An equality operator can be explicitly declared for a limited type.

Linker.

A system program that creates an executable program from one or more *object modules* (OpenVMS) or *object files* (Digital UNIX) produced by a language compiler or assembler. The linker resolves external references, acquires referenced library routines, and performs other processing required to create executable images (OpenVMS) or object files (Digital UNIX).

Literal.

A literal represents a value literally, that is, by means of letters and other characters. A literal is either a numeric literal, an enumeration literal, a character literal, or a string literal.

Membership test.

A membership test is a basic operation that indicates whether a given value is contained in a given *range* or *subtype*.

Mode.

See *parameter.*

Model number.

A model number is an exactly representable value of a *real type. Operations* of a real type are defined in terms of operations on the model numbers of the type. The properties of the model numbers and of their operations are the minimal properties preserved by all implementations of the real type.

Name.

A name is a construct that stands for an entity: it is said that the name *denotes* the entity, and that the entity is the meaning of the name. See also *declaration, prefix.*

Named association.

A named association specifies the association of an item with one or more positions in a list, by naming the positions.

Noncontiguous array.

A noncontiguous array is a *descriptor* class. A noncontiguous array is one in which the storage of the array *components* may be allocated with a fixed, nonzero number of bytes separating logically adjacent components.

Object.

An object contains a value. A program creates an object either by *elaborating* an *object declaration* or by *evaluating* an *allocator.* The declaration or allocator specifies a *type* for the object: the object can only contain values of that type.

Object file (Digital UNIX term).

The binary output of a Digital UNIX language processor (such as an assembler or compiler), which can either be executed or be used as input to the *linker*.

Object module (OpenVMS term).

The binary output of an OpenVMS language processor (such as an assembler or compiler), which is used as input to the OpenVMS *Linker*.

Operation.

An operation is an elementary action associated with one or more *types.* It is either implicitly declared by the *declaration* of the type, or it is a *subprogram* that has a *parameter* or *result* of the type.

Operator.

An operator is an operation which has one or two operands. A unary operator is written before an operand; a binary operator is written between two operands. This notation is a special kind of *function call.* An operator can be declared as a function. Many operators are implicitly declared by the *declaration* of a *type* (for example, most type declarations imply the declaration of the equality operator for values of the type).

Overloading.

An identifier can have several alternative meanings at a given point in the program text: this property is called *overloading.* For example, an overloaded enumeration literal can be an identifier that appears in the definitions of two or more *enumeration types.* The effective meaning of an overloaded identifier is determined by the context. *Subprograms*, *aggregates*, *allocators*, and string *literals* can also be overloaded.

Packable.

A *type* is packable if storage for *objects* of the type can be aligned on an arbitrary bit boundary.

Package.

A package specifies a group of logically related entities, such as *types*, *objects* of those types, and *subprograms* with *parameters* of those types. It is written as a *package declaration* and a *package body.* The package declaration has a *visible part*, containing the *declarations* of all entities that can be explicitly used outside the package. It may also have a *private part* containing structural details that complete the specification of the visible entities, but which are irrelevant to the user of the package. The *package body* contains implementations of *subprograms* (and possibly *tasks* as other *packages*) that have been specified in the package declaration. A package is one of the kinds of *program unit.*

Parameter.

A parameter is one of the named entities associated with a *subprogram*, *entry*, or *generic unit*, and used to communicate with the corresponding subprogram body, *accept statement* or generic body. A *formal parameter* is an *identifier* used to denote the named entity within the body. An *actual parameter* is the particular entity associated with the corresponding formal parameter by a *subprogram call*, *entry call*, or *generic instantiation*. The *mode* of a formal parameter specifies whether the associated actual parameter supplies a value for the formal parameter, or the formal supplies a value for the actual parameter, or both. The association of actual parameters with formal parameters can be specified by *named associations*, by *positional associations*, or by a combination of these.

Parent type.

See *derived type.*

Positional association.

A positional association specifies the association of an item with a position in a list, by using the same position in the text to specify the item.

Pragma.

A pragma conveys information to the compiler.

Prefix.

A prefix is used as the first part of certain kinds of name. A prefix is either a *function call* or a *name.*

Private part.

See *package.*

Private type.

A private type is a *type* whose structure and set of values are clearly defined, but not directly available to the user of the type. A private type is known only by its *discriminants* (if any) and by the set of *operations* defined for it. A private type and its applicable operations are defined in the *visible part* of a *package*, or in a *generic formal part. Assignment*, equality, and inequality are also defined for private types, unless the private type is *limited.*

Procedure.

See *subprogram.*

Program.

A program is composed of a number of *compilation units*, one of which is a *subprogram* called the *main program.* Execution of the program consists of execution of the main program, which may invoke subprograms declared in the other compilation units of the program.

Program section (OpenVMS term).

A program section (psect) is a portion of a *program* with a given OpenVMS protection and set of storage management attributes. Program sections with the same attributes are gathered by the OpenVMS *Linker* to form an image section.

Program unit.

A program unit is any one of a *generic unit*, *package*, *subprogram*, or *task unit.*

Psect (OpenVMS term).

See *program section.*

Qualified expression.

A qualified expression is an *expression* preceded by an indication of its *type* or *subtype.* Such qualification is used when, in its absence, the expression might be ambiguous (for example as a consequence of *overloading*).

Raising an exception.

See *exception.*

Range.

A range is a contiguous set of values of a *scalar type.* A range is specified by giving the lower and upper bounds for the values. A value in the range is said to *belong* to the range.

Range constraint.

A range constraint of a *type* specifies a *range*, and thereby determines the subset of the values of the type that *belong* to the range.

Real type.

A real type is a *type* whose values represent approximations to the real numbers. There are two kinds of real type: *fixed point types* are specified by absolute error bound; *floating point types* are specified by a relative error bound expressed as a number of significant decimal digits.

Record type.

A value of a record type consists of *components* which are usually of different *types* or *subtypes.* For each component of a record value or record *object*, the definition of the record type specifies an identifier that uniquely determines the component within the record.

Renaming declaration.

A renaming declaration declares another *name* for an entity.

Rendezvous.

A rendezvous is the interaction that occurs between two parallel *tasks* when one task has called an *entry* of the other task, and a corresponding *accept statement* is being executed by the other task on behalf of the calling task.

Representation clause.

A representation clause directs the compiler in the selection of the mapping of a *type*, an *object*, or a *task* onto features of the underlying machine that executes a program. In some cases, representation clauses completely specify the mapping; in other cases, they provide criteria for choosing a mapping.

Satisfy.

See *constraint, subtype.*

Scalar type.

An *object* or value of a scalar *type* does not have *components.* A scalar type is either a *discrete type* or a *real type.* The values of a scalar type are ordered.

Scope.

See *declaration.*

Selected component.

A selected component is a *name* consisting of a *prefix* and of an identifier called the *selector.* Selected components are used to denote record components, *entries*, and *objects* designated by access values; they are also used as *expanded names.*

Selector.

See *selected component.*

Short circuit control form.

A short circuit control form is one of the reserved word pairs **and then** and **or else**. Each has the same precedence as a logical *operator*.

Signal (Digital UNIX term).

A Digital UNIX signal is a hardware- or software-detected event that alters the normal flow of instruction execution.

Simple name.

See *declaration*, *name.*

Simple record.

A simple *record type* is one that does not have a *variant part* and in which any *constraint* for each *component* and *subcomponent* is static. A simple record *subtype* is either a simple record type or a static constrained subtype of a record type (with *discriminants*) in which any constraint for each component and subcomponent of the record type is static.

Statement.

A statement specifies one or more actions to be performed during the execution of a *program.*

Status value (OpenVMS term).

A status value is the *condition value* returned by an OpenVMS routine to indicate whether or not the routine completed successfully.

Subcomponent.

A subcomponent is either a *component*, or a component of another subcomponent.

Subprogram.

A subprogram is either a *procedure* or a *function.* A procedure specifies a sequence of actions and is invoked by a *procedure call* statement. A function specifies a sequence of actions and also returns a value called the *result*, and so a *function call* is an *expression.* A subprogram is written as a *subprogram declaration*, which specifies its *name*, *formal parameters*, and (for a function) its result; and a *subprogram body* which specifies the sequence of actions. The subprogram call specifies the *actual parameters* that are to be associated with the formal parameters. A subprogram is one of the kinds of *program unit.*

Subtype.

A subtype of a *type* characterizes a subset of the values of the type. The subset is determined by a *constraint* on the type. Each value in the set of values of a subtype *belongs* to the subtype and *satisfies* the constraint determining the subtype.

Subunit.

See *body.*

Task.

A task operates in parallel with other parts of the program. It is written as a *task specification* (which specifies the *name* of the task and the names and *formal parameters* of its entries), and a *task body* which defines its execution. A *task unit* is one of the kinds of *program unit.* A *task type* is a *type* that permits the subsequent *declaration* of any number of similar tasks of the type. A value of a task type is said to *designate* a task.

Thread.

A single, sequential flow of control within a program. Within a single thread, there is a single point of execution. Most traditional programs consist of a single thread.

Type.

A type characterizes both a set of values, and a set of *operations* applicable to those values. A *type definition* is a language construct that defines a type. A particular type is either an *access type*, an *array type*, a *private type*, a *record type*, a *scalar type*, or a *task type.*

Use clause.

A use clause achieves *direct visibility* of *declarations* that appear in the *visible parts* of named *packages.*

Variable.

See *object.*

Variant part.

A variant part of a *record* specifies alternative record *components*, depending on a *discriminant* of the record. Each value of the discriminant establishes a particular alternative of the variant part.

Visibility.

At a given point in a program text, the *declaration* of an entity with a certain identifier is said to be *visible* if the entity is an acceptable meaning for an occurrence at that point of the identifier. The declaration is *visible* by *selection* at the place of the *selector* in a *selected component* or at the place of the name in a *named association.* Otherwise, the declaration is *directly visible*, that is, if the identifier alone has that meaning.

Visible part.

See *package.*

With clause.

See *compilation unit.*

E

Syntax Summary

Note

This syntax summary is not part of the standard definition of the Ada
programming language.

2.1

```
graphic_character ::= basic_graphic_character
    | lower_case_letter | other_special_character

basic_graphic_character ::=
      upper_case_letter | digit
    | special_character | space_character

basic_character ::=
      basic_graphic_character | format_effector
```

2.3

```
identifier ::=
    letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

2.4

```
numeric_literal ::= decimal_literal | based_literal
```

2.4.1

```
decimal_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer
```

2.4.2

```
based_literal ::=
    base # based_integer [.based_integer] # [exponent]

base ::= integer

based_integer ::=
    extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter
```

2.5

```
character_literal ::= 'graphic_character'
```

2.6

```
string_literal ::= "{graphic_character}"
```

2.8

```
pragma ::=
    pragma identifier [(argument_association
                            {, argument_association})];

argument_association ::=
      [argument_identifier =>] name
    | [argument_identifier =>] expression
```

3.1

```
basic_declaration ::=
      object_declaration     | number_declaration
    | type_declaration       | subtype_declaration
    | subprogram_declaration | package_declaration
    | task_declaration       | generic_declaration
    | exception_declaration  | generic_instantiation
    | renaming_declaration   | deferred_constant_declaration
```

3.2

```
object_declaration ::=
      identifier_list : [constant] subtype_indication
                            [:= expression];
    | identifier_list : [constant] constrained_array_definition
                            [:= expression];
number_declaration ::=
      identifier_list : constant := universal_static_expression;

identifier_list ::=  identifier {, identifier}
```

3.3.1

```
type_declaration ::=  full_type_declaration
   | incomplete_type_declaration | private_type_declaration

full_type_declaration ::=
 type identifier [discriminant_part] is type_definition;

type_definition ::=
     enumeration_type_definition | integer_type_definition
   | real_type_definition      | array_type_definition
   | record_type_definition    | access_type_definition
   | derived_type_definition
```

3.3.2

```
subtype_declaration ::=
   subtype identifier is subtype_indication;

subtype_indication ::=  type_mark [constraint]

type_mark ::= type_name | subtype_name

constraint ::=
     range_constraint       | floating_point_constraint
   | fixed_point_constraint | index_constraint
   | discriminant_constraint
```

3.4

```
derived_type_definition ::= new subtype_indication
```

3.5

```
range_constraint ::=  range range

range ::=  range_attribute
   | simple_expression .. simple_expression
```

3.5.1

```
enumeration_type_definition ::=
   (enumeration_literal_specification
      {, enumeration_literal_specification})

enumeration_literal_specification ::=  enumeration_literal

enumeration_literal ::=  identifier | character_literal
```

3.5.4

```
integer_type_definition ::=  range_constraint
```

3.5.6

```
real_type_definition ::=
   floating_point_constraint | fixed_point_constraint
```

3.5.7

```
floating_point_constraint ::=
   floating_accuracy_definition [range_constraint]

floating_accuracy_definition ::=
   digits static_simple_expression
```

3.5.9

```
fixed_point_constraint ::=
   fixed_accuracy_definition [range_constraint]

fixed_accuracy_definition ::=
   delta static_simple_expression
```

3.6

```
array_type_definition ::=
      unconstrained_array_definition
    | constrained_array_definition

unconstrained_array_definition ::=
   array(index_subtype_definition
            {, index_subtype_definition}) of
            component_subtype_indication

constrained_array_definition ::=
   array index_constraint of component_subtype_indication

index_subtype_definition ::= type_mark range <>

index_constraint ::=  (discrete_range {, discrete_range})

discrete_range ::= discrete_subtype_indication | range
```

3.7

```
record_type_definition ::=
   record
      component_list
   end record

component_list ::=
      component_declaration {component_declaration}
    | {component_declaration} variant_part
    | null;

component_declaration ::=
   identifier_list : component_subtype_definition
                     [:= expression];

component_subtype_definition ::=  subtype_indication
```

3.7.1

```
discriminant_part ::=
   (discriminant_specification {; discriminant_specification})

discriminant_specification ::=
   identifier_list : type_mark [:= expression]
```

3.7.2

```
discriminant_constraint ::=
   (discriminant_association {, discriminant_association})

discriminant_association ::=
   [discriminant_simple_name {| discriminant_simple_name} =>]
             expression
```

3.7.3

```
variant_part ::=
   case discriminant_simple_name is
        variant
       {variant}
   end case;

variant ::=
   when choice {| choice} =>
      component_list

choice ::= simple_expression
   | discrete_range | others | component_simple_name
```

3.8

```
access_type_definition ::= access subtype_indication
```

3.8.1

```
incomplete_type_declaration ::=
   type identifier [discriminant_part];
```

3.9

```
declarative_part ::=
   {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::= basic_declaration
   | representation_clause | use_clause

later_declarative_item ::= body
   | subprogram_declaration | package_declaration
   | task_declaration       | generic_declaration
   | use_clause             | generic_instantiation

body ::= proper_body | body_stub
```

```
proper_body ::=
    subprogram_body | package_body | task_body
```

4.1

```
name ::= simple_name
    | character_literal  | operator_symbol
    | indexed_component  | slice
    | selected_component | attribute

simple_name ::= identifier

prefix ::= name | function_call
```

4.1.1

```
indexed_component ::= prefix(expression {, expression})
```

4.1.2

```
slice ::= prefix(discrete_range)
```

4.1.3

```
selected_component ::= prefix.selector

selector ::= simple_name
    | character_literal | operator_symbol | all
```

4.1.4

```
attribute ::= prefix'attribute_designator

attribute_designator ::=
    simple_name [(universal_static_expression)]
```

4.3

```
aggregate ::=
    (component_association {, component_association})

component_association ::=
    [choice {| choice} => ] expression
```

4.4

```
expression ::=
    relation {and relation} | relation {and then relation}
    | relation {or relation}  | relation {or else relation}
    | relation {xor relation}

relation ::=
    simple_expression [relational_operator simple_expression]
    | simple_expression [not] in range
    | simple_expression [not] in type_mark

simple_expression ::=
    [unary_adding_operator] term {binary_adding_operator term}
```

```
term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
    numeric_literal | null | aggregate | string_literal
  | name | allocator | function_call | type_conversion
  | qualified_expression | (expression)
```

4.5

```
logical_operator  ::=  and | or | xor

relational_operator  ::=  = | /= | < | <= |> | >=

binary_adding_operator  ::=  + | - | &

unary_adding_operator  ::=  + | -

multiplying_operator  ::=  * | / | mod | rem

highest_precedence_operator  ::=  ** | abs | not
```

4.6

```
type_conversion ::= type_mark(expression)
```

4.7

```
qualified_expression ::=
    type_mark'(expression) | type_mark'aggregate
```

4.8

```
allocator ::=
    new subtype_indication | new qualified_expression
```

5.1

```
sequence_of_statements ::= statement {statement}

statement ::=
    {label} simple_statement | {label} compound_statement

simple_statement ::= null_statement
    | assignment_statement | procedure_call_statement
    | exit_statement      | return_statement
    | goto_statement      | entry_call_statement
    | delay_statement     | abort_statement
    | raise_statement     | code_statement

compound_statement ::=
      if_statement     | case_statement
    | loop_statement   | block_statement
    | accept_statement | select_statement
```

```
label ::= <<label_simple_name>>

null_statement ::= null;
```

5.2
```
assignment_statement ::=
   variable_name := expression;
```

5.3
```
if_statement ::=
    if condition then
      sequence_of_statements
   {elsif condition then
      sequence_of_statements}
   [else
      sequence_of_statements]
    end if;

condition ::= boolean_expression
```

5.4
```
case_statement ::=
   case expression is
       case_statement_alternative
      {case_statement_alternative}
   end case;

case_statement_alternative ::=
   when choice {| choice } =>
      sequence_of_statements
```

5.5
```
loop_statement ::=
   [loop_simple_name:]
      [iteration_scheme] loop
         sequence_of_statements
       end loop [loop_simple_name];

iteration_scheme ::= while condition
   | for loop_parameter_specification
```

```
loop_parameter_specification ::=
    identifier in [reverse] discrete_range
```

5.6

```
block_statement ::=
    [block_simple_name:]
        [declare
            declarative_part]
         begin
            sequence_of_statements
        [exception
            exception_handler
           {exception_handler}]
         end [block_simple_name];
```

5.7

```
exit_statement ::=
    exit [loop_name] [when condition];
```

5.8

```
return_statement ::= return [expression];
```

5.9

```
goto_statement ::= goto label_name;
```

6.1

```
subprogram_declaration ::= subprogram_specification;

subprogram_specification ::=
      procedure identifier [formal_part]
    | function designator  [formal_part] return type_mark

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

formal_part ::=
    (parameter_specification {; parameter_specification})

parameter_specification ::=
    identifier_list : mode type_mark [:= expression]
```

```
  mode ::= [in] | in out | out
```

6.3
```
subprogram_body ::=
    subprogram_specification is
       [declarative_part]
    begin
        sequence_of_statements
   [exception
        exception_handler
        {exception_handler}]
    end [designator];
```

6.4
```
procedure_call_statement ::=
   procedure_name [actual_parameter_part];

function_call ::=
   function_name [actual_parameter_part]

actual_parameter_part ::=
   (parameter_association {, parameter_association})

parameter_association ::=
   [formal_parameter =>] actual_parameter

formal_parameter ::= parameter_simple_name

actual_parameter ::=
   expression | variable_name | type_mark(variable_name)
```

7.1
```
package_declaration ::= package_specification;

package_specification ::=
    package identifier is
       {basic_declarative_item}
   [private
       {basic_declarative_item}]
    end [package_simple_name]

package_body ::=
    package body package_simple_name is
       [declarative_part]
   [begin
        sequence_of_statements
   [exception
        exception_handler
        {exception_handler}]]
    end [package_simple_name];
```

7.4
```
private_type_declaration ::=
   type identifier [discriminant_part] is [limited] private;
```

```
deferred_constant_declaration ::=
    identifier_list : constant type_mark;
```

8.4

```
use_clause ::= use package_name {, package_name};
```

8.5

```
renaming_declaration ::=
      identifier : type_mark     renames object_name;
    | identifier : exception     renames exception_name;
    | package identifier         renames package_name;
    | subprogram_specification renames subprogram_or_entry_name;
```

9.1

```
task_declaration ::= task_specification;

task_specification ::=
    task [type] identifier [is
        {entry_declaration}
        {representation_clause}
    end [task_simple_name]]

task_body ::=
     task body task_simple_name is
         [declarative_part]
     begin
         sequence_of_statements
    [exception
         exception_handler
         {exception_handler}]
     end [task_simple_name];
```

9.5

```
entry_declaration ::=
    entry identifier [(discrete_range)] [formal_part];

entry_call_statement ::=
    entry_name [actual_parameter_part];

accept_statement ::=
    accept entry_simple_name [(entry_index)] [formal_part] [do
        sequence_of_statements
    end [entry_simple_name]];

entry_index ::= expression
```

9.6

```
delay_statement ::= delay simple_expression;
```

9.7

```
select_statement ::= selective_wait
    | conditional_entry_call | timed_entry_call
```

9.7.1

```
selective_wait ::=
     select
        select_alternative
    {or
        select_alternative}
    [else
        sequence_of_statements]
     end select;
select_alternative ::=
    [when condition =>]
        selective_wait_alternative
selective_wait_alternative ::= accept_alternative
    | delay_alternative | terminate_alternative
accept_alternative  ::=
    accept_statement [sequence_of_statements]
delay_alternative  ::=
    delay_statement [sequence_of_statements]
terminate_alternative ::= terminate;
```

9.7.2

```
conditional_entry_call ::=
    select
        entry_call_statement
        [sequence_of_statements]
    else
        sequence_of_statements
    end select;
```

9.7.3

```
timed_entry_call ::=
    select
        entry_call_statement
        [sequence_of_statements]
    or
        delay_alternative
    end select;
```

9.10

```
abort_statement ::= abort task_name {, task_name};
```

10.1

```
compilation ::= {compilation_unit}

compilation_unit ::=
     context_clause library_unit
   | context_clause secondary_unit

library_unit ::=
     subprogram_declaration | package_declaration
   | generic_declaration    | generic_instantiation
   | subprogram_body

secondary_unit ::= library_unit_body | subunit

library_unit_body ::= subprogram_body | package_body
```

10.1.1

```
context_clause ::= {with_clause {use_clause}}

with_clause ::=
   with unit_simple_name {, unit_simple_name};
```

10.2

```
body_stub ::=
     subprogram_specification is separate;
   | package body package_simple_name is separate;
   | task body task_simple_name is separate;

subunit ::= separate (parent_unit_name) proper_body
```

11.1

```
exception_declaration ::= identifier_list : exception;
```

11.2

```
exception_handler ::=
   when exception_choice {| exception_choice} =>
      sequence_of_statements

exception_choice ::= exception_name | others
```

11.3

```
raise_statement ::= raise [exception_name];
```

12.1

```
generic_declaration ::= generic_specification;

generic_specification ::=
     generic_formal_part subprogram_specification
   | generic_formal_part package_specification
```

```
generic_formal_part ::=
    generic {generic_parameter_declaration}

generic_parameter_declaration ::=
      identifier_list : [in [out]] type_mark [:= expression];
    | type identifier is generic_type_definition;
    | private_type_declaration
    | with subprogram_specification [is name];
    | with subprogram_specification [is <>];

generic_type_definition ::=
      (<>) | range <> | digits <> | delta <>
    | array_type_definition | access_type_definition
```

12.3

```
generic_instantiation ::=
      package identifier is
          new generic_package_name [generic_actual_part];
    | procedure identifier is
          new generic_procedure_name [generic_actual_part];
    | function designator is
          new generic_function_name [generic_actual_part];

generic_actual_part ::=
    (generic_association {, generic_association})

generic_association ::=
    [generic_formal_parameter =>] generic_actual_parameter

generic_formal_parameter ::=
    parameter_simple_name | operator_symbol

generic_actual_parameter ::= expression | variable_name
    | subprogram_name | entry_name | type_mark
```

13.1

```
representation_clause ::=
      type_representation_clause | address_clause

type_representation_clause ::= length_clause
    | enumeration_representation_clause
    | record_representation_clause
```

13.2

```
length_clause ::= for attribute use simple_expression;
```

13.1a

```
alignment_choice ::=
      COMPONENT_SIZE | COMPONENT_SIZE_4
    | DEFAULT        | STORAGE_UNIT
```

13.2b

```
main_storage_option ::=
      [WORKING_STORAGE =>] static_simple_expression
    | [TOP_GUARD =>] static_simple_expression
```

13.3

```
enumeration_representation_clause ::=
    for type_simple_name use aggregate;
```

13.4

```
record_representation_clause ::=
    for type_simple_name use
       record [alignment_clause]
          {component_clause}
       end record;

alignment_clause ::= at mod static_simple_expression;

component_clause ::=
    component_name at static_simple_expression
                 range static_range;
```

13.5

```
address_clause ::=
    for simple_name use at simple_expression;
```

13.8

```
code_statement ::= type_mark'record_aggregate;
```

13.9a

```
external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal

import_export_pragma_name ::=
      COMMON_OBJECT
    | EXPORT_EXCEPTION   -- OpenVMS systems only
    | EXPORT_FUNCTION  | EXPORT_OBJECT
    | EXPORT_PROCEDURE | EXPORT_VALUED_PROCEDURE
    | IMPORT_EXCEPTION   -- OpenVMS systems only
    | IMPORT_FUNCTION  | IMPORT_OBJECT
    | IMPORT_PROCEDURE | IMPORT_VALUED_PROCEDURE
    | PSECT_OBJECT       -- OpenVMS systems only

internal_name ::=
      [INTERNAL =>] simple_name
    | [INTERNAL =>] operator_symbol -- EXPORT_FUNCTION and
                                    -- IMPORT_FUNCTION only
```

13.9a.1.1

```
class_name ::= UBS | UBSB | UBA | S | SB | A | NCA

mechanism ::=
    mechanism_name | (mechanism_association {, mechanism_association })

mechanism_association ::=
    [formal_parameter =>] mechanism_name

mechanism_name ::=
      VALUE
    | REFERENCE
    | DESCRIPTOR [([CLASS =>] class_name)] -- OpenVMS
                                          -- systems only
parameter_types ::= null | type_mark {, type_mark}
```

13.9b (pragma INTERFACE_NAME)

```
external_name ::= string_literal
```

B (pragma TITLE)

```
titling_option ::=
      [TITLE =>] string_literal
    | [SUBTITLE =>] string_literal
```

Syntax Cross Reference

In the list given below each syntactic category is followed by the section
number where it is defined. For example:

adding_operator   4.5

In addition, each syntactic category is followed by the names of other categories
in whose definition it appears. For example, adding_operator appears in the
definition of simple_expression:

adding_operator   4.5
    simple_expression   4.4

An ellipsis (. . .) is used when the syntactic category is not defined by a syntax
rule. For example:

lower_case_letter   . . .

All uses of parentheses are combined in the term "()". The italicized prefixes
used with some terms have been deleted here.

abort   . . .
    abort_statement   9.10

>>   . . .
    label   **5.1**

| . . .
    case_statement_alternative   **5.4**
    component_association   **4.3**
    discriminant_association   **3.7.2**
    exception_handler   **11.2**
    variant   **3.7.3**

F

Implementation-Dependent Characteristics

> **Note**
>
> This appendix is not part of the standard definition of the Ada programming language.

This appendix summarizes the implementation-dependent characteristics of DEC Ada by presenting the following:

- Lists of the DEC Ada pragmas and attributes
- The specification of the package SYSTEM
- The restrictions on representation clauses and unchecked type conversions
- The conventions for names denoting implementation-dependent components in record representation clauses
- The interpretation of expressions in address clauses
- The implementation-dependent characteristics of the input-output packages
- Other implementation-dependent characteristics

See the relevant run-time reference manual for additional implementation-specific details.

F.1 Implementation-Dependent Pragmas

DEC Ada provides the following pragmas, which are defined elsewhere in the text:

Pragma	DEC Ada systems on which it applies	Section
AST_ENTRY	OpenVMS	9.12a
COMMON_OBJECT	All	13.9a.2.3
COMPONENT_ALIGNMENT	All	13.1a
EXPORT_EXCEPTION	OpenVMS	13.9a.3.2
EXPORT_FUNCTION	All	13.9a.1.3
EXPORT_OBJECT	All	13.9a.2.2
EXPORT_PROCEDURE	All	13.9a.1.3
EXPORT_VALUED_PROCEDURE	All	13.9a.1.3
FLOAT_REPRESENTATION	All	3.5.7a
IDENT	All	Annex B
IMPORT_EXCEPTION	OpenVMS	13.9a.3.1
IMPORT_FUNCTION	All	13.9a.1.1
IMPORT_OBJECT	All	13.9a.2.1
IMPORT_PROCEDURE	All	13.9a.1.1
IMPORT_VALUED_PROCEDURE	All	13.9a.1.1
INLINE_GENERIC	All	12.1a
INTERFACE_NAME	All	13.9b
LONG_FLOAT	OpenVMS	3.5.7b
MAIN_STORAGE	OpenVMS VAX	13.2b
PSECT_OBJECT	OpenVMS	13.9a.2.3
PASSIVE	OpenVMS Alpha Digital UNIX	9.8b
SHARE_GENERIC	All	12.1b
SUPPRESS_ALL	All	11.7
TASK_STORAGE	All	13.2a
TIME_SLICE	OpenVMS	9.8a
TITLE	All	Annex B
VOLATILE	All	9.11

DEC Ada also restricts the predefined language pragmas INLINE and INTERFACE. See Annex B for a descriptive pragma summary.

F.2 Implementation-Dependent Attributes

DEC Ada provides the following attributes, which are defined elsewhere in the text:

Attribute	DEC Ada systems on which it applies	Section
AST_ENTRY	OpenVMS	9.12a
BIT	All	13.7.2
MACHINE_SIZE	All	13.7.2
NULL_PARAMETER	All	13.9a.1.2
TYPE_CLASS	All	13.7a.2

See Annex A for a descriptive attribute summary.

F.3 Specification of the Package System

DEC Ada provides a system-specific version of the package SYSTEM for each system on which it is supported. The following specification is composite: it includes all common and system-specific features.

```
package SYSTEM is

    type NAME is
        -- DEC Ada implementations
      (VAX_VMS, OpenVMS_AXP, RISC_ULTRIX, DEC_OSF1_AXP,
        -- XD Ada implementations
       MIL_STD_1750A, MC68000, MC68020, MC68040, CPU32);
    for NAME use (1, 2, 7, 8, 9, 101, 102, 103, 104, 105);

    -- System-specific definitions of SYSTEM_NAME
    SYSTEM_NAME   : constant NAME := VAX_VMS;
    SYSTEM_NAME   : constant NAME := OpenVMS_AXP;
    SYSTEM_NAME   : constant NAME := DEC_OSF1_AXP;
    SYSTEM_NAME   : constant NAME := RISC_ULTRIX;

    STORAGE_UNIT  : constant := 8;
    MEMORY_SIZE   : constant := 2**63-1;
    MEMORY_SIZE   : constant := 2**31-1;
```

```
-- System-specific definitions of MAX_INT and MIN_INT
--
---- On Alpha systems:
MAX_INT       : constant := 2**63-1;
MIN_INT       : constant := -(2**63);
--
---- On all other systems:
MAX_INT       : constant := 2**31-1;
MIN_INT       : constant := -(2**31);

-- System-specific definitions of MAX_DIGITS
--
---- On OpenVMS VAX systems:
MAX_DIGITS    : constant := 33;
--
---- On all other systems:
MAX_DIGITS    : constant := 15;

MAX_MANTISSA  : constant := 31;
FINE_DELTA    : constant := 2.0**(-31);


-- System-specific definitions of TICK
--
---- On OpenVMS VAX systems:
TICK          : constant := 10.0**(-2);
--
---- On Alpha systems:
TICK          : constant := 10.0**(-3);
--

subtype PRIORITY is INTEGER range 0 .. 15;

-- On OpenVMS and Digital UNIX systems:
type INTEGER_8 is range -128 .. 127;
for INTEGER_8'SIZE use 8;

-- On OpenVMS and Digital UNIX systems:
type INTEGER_16 is range -32_768 .. 32_767;
for INTEGER_16'SIZE use 16;

-- On OpenVMS and Digital UNIX systems:
type INTEGER_32 is range -2_147_483_648 .. 2_147_483_647;
for INTEGER_32'SIZE use 32;

-- On Alpha systems:
type INTEGER_64 is range
   -9_223_372_036_854_775_808 .. 9_223_372_036_854_775_807;
for INTEGER_64'SIZE use 64;

-- On OpenVMS and Digital UNIX systems:
type LARGEST_INTEGER is range MIN_INT .. MAX_INT;
```

```
      -- On OpenVMS and Digital UNIX systems:
      function "not" (LEFT        : LARGEST_INTEGER) return LARGEST_INTEGER;
      function "and" (LEFT, RIGHT : LARGEST_INTEGER) return LARGEST_INTEGER;
      function "or"  (LEFT, RIGHT : LARGEST_INTEGER) return LARGEST_INTEGER;
      function "xor" (LEFT, RIGHT : LARGEST_INTEGER) return LARGEST_INTEGER;

-- Address type
--
      type ADDRESS is private;

      ADDRESS_ZERO : constant ADDRESS;
      NO_ADDR      : constant ADDRESS;
      NULL_ADDRESS : constant ADDRESS;

      -- System-specific definition of ADDRESS_SIZE
      --
      ---- On Digital UNIX systems:
      ADDRESS_SIZE : constant := 64;
      --
      ---- On OpenVMS systems:
      ADDRESS_SIZE : constant := 32;

      function "+" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
      function "+" (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
      function "-" (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
      function "-" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
-- Note that because ADDRESS is a private type
-- the functions "=" and "/=" are already available and
-- do not have to be explicitly defined
--
-- function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
      function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
      function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
      function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
      function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

      generic
         type TARGET is private;
      function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

      generic
         type TARGET is private;
      procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);

-- DEC Ada floating point type declarations for the VAX
-- floating point data types
```

```
      --
      ---- On OpenVMS systems:
      type F_FLOAT is {digits 6};
      type D_FLOAT is {digits 9};
      type G_FLOAT is {digits 15};
      --
      ---- On OpenVMS VAX systems:
      type H_FLOAT is {digits 33};
   -- DEC Ada floating point type declarations for the IEEE
   -- floating point data types

      --
      ---- On Alpha systems:
      type IEEE_SINGLE_FLOAT is {digits 6};
      type IEEE_DOUBLE_FLOAT is {digits 15};

      type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                          TYPE_CLASS_INTEGER,
                          TYPE_CLASS_FIXED_POINT,
                          TYPE_CLASS_FLOATING_POINT,
                          TYPE_CLASS_ARRAY,
                          TYPE_CLASS_RECORD,
                          TYPE_CLASS_ACCESS,
                          TYPE_CLASS_TASK,
                          TYPE_CLASS_ADDRESS);

   -- AST handler type

      -- On OpenVMS systems:
      type AST_HANDLER is limited private;

      -- On OpenVMS systems:
      NO_AST_HANDLER : constant AST_HANDLER;

   -- Non-Ada exception

      NON_ADA_ERROR : exception;

   -- Hardware-oriented types and functions

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

      subtype BIT_ARRAY_8  is BIT_ARRAY (0 ..  7);
      subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
      subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
      subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

      type UNSIGNED_BYTE is range 0 .. 255;
      for  UNSIGNED_BYTE'SIZE use 8;

      function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
      function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
      function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
      function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
```

```
function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
function TO_BIT_ARRAY_8   (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

type UNSIGNED_WORD is range 0 .. 65535;
for  UNSIGNED_WORD'SIZE use 16;

function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
function TO_BIT_ARRAY_16  (X : UNSIGNED_WORD) return BIT_ARRAY_16;

type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

type UNSIGNED_LONGWORD is range -2_147_483_648 .. 2_147_483_647;
for  UNSIGNED_LONGWORD'SIZE use 32;

function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32) return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

type UNSIGNED_LONGWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_LONGWORD;

-- On Alpha systems:
type UNSIGNED_32 is range 0 .. 4_294_967_295;
for  UNSIGNED_32'SIZE use 32;

-- On Alpha systems:
function "not" (LEFT        : UNSIGNED_32) return UNSIGNED_32;
function "and" (LEFT, RIGHT : UNSIGNED_32) return UNSIGNED_32;
function "or"  (LEFT, RIGHT : UNSIGNED_32) return UNSIGNED_32;
function "xor" (LEFT, RIGHT : UNSIGNED_32) return UNSIGNED_32;

-- On Alpha systems:
function TO_UNSIGNED_32 (X : BIT_ARRAY_32) return UNSIGNED_32;
function TO_BIT_ARRAY_32 (X : UNSIGNED_32) return BIT_ARRAY_32;

type UNSIGNED_QUADWORD is
   record
      L0 : UNSIGNED_LONGWORD;
      L1 : UNSIGNED_LONGWORD;
   end record;
for  UNSIGNED_QUADWORD'SIZE use 64;
for UNSIGNED_QUADWORD use
   record at mod 8;
   end record;
```

```
    function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
    function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

    function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64) return UNSIGNED_QUADWORD;
    function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;

    type UNSIGNED_QUADWORD_ARRAY is
       array (INTEGER range <>) of UNSIGNED_QUADWORD;

    function TO_ADDRESS  (X : INTEGER)             return ADDRESS;
    function TO_ADDRESS  (X : UNSIGNED_LONGWORD)   return ADDRESS;
    function TO_ADDRESS  (X : {universal_integer}) return ADDRESS;

    function TO_INTEGER          (X : ADDRESS)     return INTEGER;
    function TO_UNSIGNED_LONGWORD (X : ADDRESS)    return UNSIGNED_LONGWORD;

    -- On OpenVMS systems only:
    function TO_UNSIGNED_LONGWORD (X : AST_HANDLER) return UNSIGNED_LONGWORD;

-- Conventional names for static subtypes of type UNSIGNED_LONGWORD

    subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
    subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
    subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
    subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
    subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
    subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
    subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
    subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
    subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
    subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
    subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
    subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
    subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
    subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
    subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
    subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
    subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
    subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
    subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
    subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
    subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
    subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
    subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
    subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
    subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
    subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
    subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
    subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
    subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
    subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
    subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;
```

```
-- Function for obtaining global symbol values

   function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;

   -- On OpenVMS and Digital UNIX systems:
   function IMPORT_ADDRESS (SYMBOL : STRING) return ADDRESS;
   function IMPORT_LARGEST_VALUE (SYMBOL : STRING) return LARGEST_INTEGER;

-- VAX device and process register operations

   -- On OpenVMS VAX systems only:
   function  READ_REGISTER (SOURCE : UNSIGNED_BYTE) return UNSIGNED_BYTE;
   function  READ_REGISTER (SOURCE : UNSIGNED_WORD) return UNSIGNED_WORD;
   function  READ_REGISTER (SOURCE : UNSIGNED_LONGWORD)
      return UNSIGNED_LONGWORD;

   -- On OpenVMS VAX systems only:
   procedure WRITE_REGISTER(SOURCE : UNSIGNED_BYTE;
                            TARGET : out UNSIGNED_BYTE);
   procedure WRITE_REGISTER(SOURCE : UNSIGNED_WORD;
                            TARGET : out UNSIGNED_WORD);
   procedure WRITE_REGISTER(SOURCE : UNSIGNED_LONGWORD;
                            TARGET : out UNSIGNED_LONGWORD);

   -- On OpenVMS VAX systems only:
   function  MFPR (REG_NUMBER : INTEGER) return UNSIGNED_LONGWORD;
   procedure MTPR (REG_NUMBER : INTEGER;
                   SOURCE     : UNSIGNED_LONGWORD);

   -- For the following declarations, note that the declaration without
   -- a RETRY_COUNT parameter mean to retry infinitely.  A value of 0
   -- for the RETRY_COUNT means do not retry.
   --
-- Interlocked-instruction procedures

   -- On OpenVMS and Digital UNIX systems:
   procedure CLEAR_INTERLOCKED (BIT       : in out BOOLEAN;
                                OLD_VALUE : out BOOLEAN);
   procedure SET_INTERLOCKED   (BIT       : in out BOOLEAN;
                                OLD_VALUE : out BOOLEAN);

   -- On OpenVMS and Digital UNIX systems:
   type ALIGNED_WORD is
      record
         VALUE : SHORT_INTEGER;
      end record;
   for  ALIGNED_WORD use
      record at mod 2;
      end record;
```

```ada
-- On Alpha systems only:
procedure CLEAR_INTERLOCKED (BIT          : in out BOOLEAN;
                             OLD_VALUE    : out BOOLEAN;
                             RETRY_COUNT  : in NATURAL;
                             SUCCESS_FLAG : out BOOLEAN);
procedure SET_INTERLOCKED   (BIT          : in out BOOLEAN;
                             OLD_VALUE    : out BOOLEAN;
                             RETRY_COUNT  : in NATURAL;
                             SUCCESS_FLAG : out BOOLEAN);

-- On OpenVMS and Digital UNIX systems:
procedure ADD_INTERLOCKED (ADDEND : in     SHORT_INTEGER;
                           AUGEND : in out ALIGNED_WORD;
                           SIGN   : out    INTEGER);

-- On Alpha systems only:
type ALIGNED_INTEGER is
   record
      VALUE : INTEGER;
   end record;
for  ALIGNED_INTEGER use
   record at mod 4;
   end record;

-- On Alpha systems only:
type ALIGNED_LONG_INTEGER is
   record
      VALUE : LONG_INTEGER;
   end record;
for  ALIGNED_LONG_INTEGER use
   record at mod 8;
   end record;

-- For the following declarations, note that the declaration without
-- a RETRY_COUNT parameter mean to retry infinitely.  A value of 0
-- for the RETRY_COUNT means do not retry.
--
-- On Alpha systems only:
procedure ADD_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      AMOUNT       : in INTEGER);
procedure ADD_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      AMOUNT       : in INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
procedure ADD_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      AMOUNT       : in LONG_INTEGER);
procedure ADD_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      AMOUNT       : in LONG_INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out LONG_INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
```

```ada
-- On Alpha systems only:
procedure AND_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      FROM         : in INTEGER);
procedure AND_ATOMIC (TO           : in out ALIGNED_INTEGER;
                      FROM         : in INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
procedure AND_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      FROM         : in LONG_INTEGER);
procedure AND_ATOMIC (TO           : in out ALIGNED_LONG_INTEGER;
                      FROM         : in LONG_INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out LONG_INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);

-- On Alpha systems only:
procedure OR_ATOMIC  (TO           : in out ALIGNED_INTEGER;
                      FROM         : in INTEGER);
procedure OR_ATOMIC  (TO           : in out ALIGNED_INTEGER;
                      FROM         : in INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);
procedure OR_ATOMIC  (TO           : in out ALIGNED_LONG_INTEGER;
                      FROM         : in LONG_INTEGER);
procedure OR_ATOMIC  (TO           : in out ALIGNED_LONG_INTEGER;
                      FROM         : in LONG_INTEGER;
                      RETRY_COUNT  : in NATURAL;
                      OLD_VALUE    : out LONG_INTEGER;
                      SUCCESS_FLAG : out BOOLEAN);

-- On OpenVMS VAX systems only:
type INSQ_STATUS is (OK_NOT_FIRST, FAIL_NO_LOCK, OK_FIRST);
for INSQ_STATUS use (OK_NOT_FIRST  => 0,
                     FAIL_NO_LOCK  => 1,
                     OK_FIRST      => 2);

-- On OpenVMS VAX systems only:
type REMQ_STATUS is (OK_NOT_EMPTY,  FAIL_NO_LOCK,
                     OK_EMPTY,      FAIL_WAS_EMPTY);
for REMQ_STATUS use (OK_NOT_EMPTY   => 0,
                     FAIL_NO_LOCK   => 1,
                     OK_EMPTY       => 2,
                     FAIL_WAS_EMPTY => 3);

-- On OpenVMS Alpha systems only:
type INSQ_STATUS is (FAIL_NO_LOCK, OK_NOT_FIRST, OK_FIRST);
for INSQ_STATUS use (FAIL_NO_LOCK  => -1,
                     OK_NOT_FIRST  => 0,
                     OK_FIRST      => 1);
```

```
      -- On OpenVMS Alpha systems only:
      type REMQ_STATUS is (FAIL_NO_LOCK,  FAIL_WAS_EMPTY,
                           OK_NOT_EMPTY,  OK_EMPTY);
      for REMQ_STATUS use (FAIL_NO_LOCK   => -1,
                           FAIL_WAS_EMPTY => 0,
                           OK_NOT_EMPTY   => 1,
                           OK_EMPTY       => 2);

      -- On OpenVMS systems only:
      procedure INSQHI (ITEM   : in  ADDRESS;
                        HEADER : in  ADDRESS;
                        STATUS : out INSQ_STATUS);

      -- On OpenVMS systems only:
      procedure REMQHI (HEADER : in  ADDRESS;
                        ITEM   : out ADDRESS;
                        STATUS : out REMQ_STATUS);

      -- On OpenVMS systems only:
      procedure INSQTI (ITEM   : in  ADDRESS;
                        HEADER : in  ADDRESS;
                        STATUS : out INSQ_STATUS);

      -- On OpenVMS systems only:
      procedure REMQTI (HEADER : in  ADDRESS;
                        ITEM   : out ADDRESS;
                        STATUS : out REMQ_STATUS);

   private

      -- Not shown

   end SYSTEM;
```

F.4 Restrictions on Representation Clauses

The representation clauses allowed in DEC Ada are length, enumeration, record representation, and address clauses.

In DEC Ada, a representation clause is not allowed for:

- A generic formal type

- A type that depends on a generic formal type

- A composite type that has a component or subcomponent of a generic formal type

- A type derived from a generic formal type

F.5 Restrictions on Unchecked Type Conversions

DEC Ada supports the generic function UNCHECKED_CONVERSION with the following restrictions on the class of types involved:

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type.

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained type with discriminants.

When the target type is a type with discriminants, the value resulting from a call of the conversion function resulting from an instantiation of UNCHECKED_CONVERSION is checked to ensure that the discriminants satisfy the constraints of the actual subtype.

If the size of the source value is greater than the size of the target subtype, the high order bits of the value are ignored (truncated). If the size of the source value is less than the size of the target subtype, the value is extended with zero bits to form the result value.

F.6 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components in Record Representation Clauses

DEC Ada does not allocate implementation-dependent components in records.

F.7 Interpretation of Expressions Appearing in Address Clauses

Expressions appearing in address clauses must be of the type ADDRESS defined in the package SYSTEM (see 13.7a.1 and F.3). In DEC Ada, values of the type SYSTEM.ADDRESS are interpreted as virtual addresses in the machine's address space.

DEC Ada allows address clauses for objects and imported subprograms. On Digital UNIX systems, the simple name can also be the name of a single entry (see 13.5).

On OpenVMS systems, DEC Ada does not support interrupts as defined in section 13.5.1. Instead, DEC Ada provides the pragma AST_ENTRY and the AST_ENTRY attribute as alternative mechanisms for handling asynchronous interrupts from the OpenVMS operating system (see 9.12a).

On Digital UNIX systems, DEC Ada supports interrupts as defined in section 13.5.1, to allow some Digital UNIX signals to be associated with task entry calls. For information on handling Digital UNIX signals, see the *DEC Ada Run-Time Reference Manual for DEC OSF/1 Systems*.

F.8 Implementation-Dependent Characteristics of Input-Output Packages

In addition to the standard predefined input-output packages (SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, and IO_EXCEPTIONS), DEC Ada provides the following packages for handling sequential and direct files with mixed-type elements:

- SEQUENTIAL_MIXED_IO (see 14.2b.4)

- DIRECT_MIXED_IO (see 14.2b.6)

DEC Ada does not provide the low level input-output package described in this section.

As specified in section 14.4, DEC Ada raises the following language-defined exceptions for error conditions that occur during input-output operations:

- STATUS_ERROR

- MODE_ERROR

- NAME_ERROR

- USE_ERROR

- END_ERROR

- DATA_ERROR

- LAYOUT_ERROR

DEC Ada does not raise the language-defined exception DEVICE_ERROR. Device-related errors cause the exception USE_ERROR to be raised.

The exception USE_ERROR is raised under the following conditions:

- If the capacity of the external file has been exceeded

- In all CREATE operations if the mode specified is IN_FILE

- In all CREATE operations if the file attributes specified by the FORM parameter are not supported by the package

- In all CREATE, OPEN, DELETE, and RESET operations if, for the specified mode, the environment does not support the operation for an external file

- In all NAME operations if the file has no name

- In the SET_LINE_LENGTH and SET_PAGE_LENGTH operations on text files if the lengths specified are inappropriate for the external file

- In text files if an operation is attempted that is not possible for reasons that depend on characteristics of the external file

DEC Ada provides other input-output packages that are available on specific systems. The following sections outline those packages. The following sections also give system-specific information about the overall set of DEC Ada input-output packages and input-output exceptions.

F.8.1 DEC Ada Input-Output Packages on OpenVMS Systems

On OpenVMS systems, the DEC Ada predefined packages and their operations are implemented using OpenVMS Record Management Services (RMS) file organizations and facilities. To give users the maximum benefit of the underlying RMS input-output facilities, DEC Ada provides the following OpenVMS-specific packages:

- RELATIVE_IO (see 14.2a.3)

- INDEXED_IO (see 14.2a.5)

- RELATIVE_MIXED_IO (see 14.2b.8)

- INDEXED_MIXED_IO (see 14.2b.10)

- AUX_IO_EXCEPTIONS (see 14.5a)

The following sections summarize the implementation-dependent characteristics of the DEC Ada input-output packages. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* discusses these characteristics in more detail.

F.8.1.1 Interpretation of the FORM Parameter on OpenVMS Systems

On OpenVMS systems, the value of the FORM parameter can be a string of statements of the OpenVMS Record Management Services (RMS) File Definition Language (FDL), or it can be a string referring to a text file of FDL statements (called an FDL file).

FDL is a special-purpose OpenVMS language for writing file specifications. These specifications are then used by DEC Ada run-time routines to create or open files. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for the rules governing the FORM parameter and for a general

description of FDL. See the *Guide to OpenVMS File Applications* and the *OpenVMS Record Management Utilities Reference Manual* for complete information on FDL.

On OpenVMS systems, each input-output package has a default string of FDL statements that is used to open or create a file. In general, specification of a FORM parameter is not necessary: it is never necessary in an OPEN procedure; it may be necessary in a CREATE procedure. The packages for which a value for the FORM parameter must be specified in a CREATE procedure are as follows:

- The packages DIRECT_IO and RELATIVE_IO require that a maximum element (record) size be specified in the FORM parameter if the item with which the package is instantiated is unconstrained.

- The packages DIRECT_MIXED_IO and RELATIVE_MIXED_IO require that a maximum element (record) size be specified in the FORM parameter.

- The packages INDEXED_IO and INDEXED_MIXED_IO require that information about keys be specified in the FORM parameter.

Any explicit FORM specification supersedes the default attributes of the governing input-output package. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* describes the default external file attributes of each input-output package.

The use of the FORM parameter is described for each input-output package in chapter 14. For information on the default FORM parameters for each DEC Ada input-output package and for information on using the FORM parameter to specify external file attributes, see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems*. For information on FDL, see the *Guide to OpenVMS File Applications* and the *OpenVMS Record Management Utilities Reference Manual*.

F.8.1.2 Input-Output Exceptions on OpenVMS Systems

In addition to the DEC Ada exceptions that apply on all systems, the following also apply on OpenVMS systems:

- The DEC Ada exceptions LOCK_ERROR, EXISTENCE_ERROR, and KEY_ERROR are raised for relative and indexed input-output operations.

- The exception USE_ERROR is raised as follows in relative and indexed files:

  – In the WRITE operations on relative or indexed files if the element in the position indicated has already been written

- In the DELETE_ELEMENT operations on relative and indexed files if the current element is undefined at the start of the operation

- In the UPDATE operations on indexed files if the current element is undefined or if the specified key violates the external file attributes

- The exception NAME_ERROR is raised as specified in section 14.4: by a call of a CREATE or OPEN procedure if the string given for the NAME parameter does not allow the identification of an external file. On OpenVMS systems, the value of a NAME parameter can be a string that denotes an OpenVMS file specification or an OpenVMS logical name (in either case, the string names an external file).

  For a CREATE procedure, the value of a NAME parameter can also be a null string, in which case it names a temporary external file that is deleted when the main program exits. The *DEC Ada Run-Time Reference Manual for OpenVMS Systems* explains the naming of external files in more detail.

- The exception LAYOUT_ERROR is raised as specified in section 14.4: in text input-output by the functions COL, LINE, or PAGE if the value returned exceeds COUNT'LAST. The exception LAYOUT_ERROR is also raised on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths and by attempts to output too many characters to a string.

  In the DEC Ada mixed input-output packages, the exception LAYOUT_ERROR is raised by the procedure GET_ITEM if no more items can be read from the file buffer. It is raised by the procedure PUT_ITEM if the current position exceeds the file buffer size.

F.8.2 Input-Output Packages on Digital UNIX Systems

On Digital UNIX systems, the DEC Ada predefined packages and their operations are implemented using Digital UNIX file facilities. DEC Ada provides no additional input-output packages specifically related to Digital UNIX systems.

The following sections summarize the Digital UNIX-specific characteristics of the DEC Ada input-output packages.

F.8.2.1 Interpretation of the FORM Parameter on Digital UNIX Systems

On Digital UNIX systems, the value of the FORM parameter conforms to the description of the FORM parameter in IEEE Standard 1003.5-1992, *IEEE Standard for Information Technology, POSIX Ada Language Interfaces*. The value of the FORM parameter must be a character string, defined as follows:

```
form_image ::=
   [field_name_identifier => field_value
      {,field_name_identifier => field _value}]

field_value ::= field_value_identifier | integer
```

The field names and field values supported by DEC Ada are as follows:

OWNER, GROUP, OTHER
Determines the file permissions associated with the file.

The field value can be NONE, READ, WRITE, EXECUTE, or any combination of the latter three values separated by underscores (for example, READ_WRITE). The field values set the access permissions for the created file. If no field value is specified, the default value is READ_WRITE_EXECUTE.

The file permissions field names and values can be used only in the FORM parameter of a CREATE procedure. The exception USE_ERROR is raised if they are used in the FORM parameter of an OPEN procedure.

APPEND
Determines whether or not data can be appended to the file.

The field value must be either TRUE or FALSE. A value of TRUE causes any output to be written to the end of the named external file. If no field value is specified, the default value is FALSE.

The append field name and value can be used only in the FORM parameter of an OPEN procedure. The exception USE_ERROR is raised if they are used in the FORM parameter of a CREATE procedure.

BLOCKING
Determines whether or not blocking input-output is in effect.

	The field value must be TASKS. This value causes a calling task (but not other tasks) to wait for the completion of any input-output operation on the file.
TERMINAL_INPUT	Specifies how characters are read from the keyboard. This field applies to text files only.
	The field value must be either LINES or CHARACTERS. The value LINES causes the canonical terminal input. The value CHARACTERS causes noncanonical terminal input. If no field value is specified, the default value is LINES.
	The terminal input field name and value have no effect if the file is not opened for input or if the file is not opened on a terminal.
FILE_STRUCTURE	Determines whether the file is first-in first-out (FIFO) or not.
	The field value must be either REGULAR or FIFO. If no field value is specified, the default value is REGULAR.
	The file structure field name and values may be used only in the FORM parameter of a CREATE procedure. The exception USE_ERROR is raised if they are used in the FORM parameter of an OPEN procedure.
	The exception USE_ERROR is also raised if the field name and values are applied to files created or opened with operations from the packages DIRECT_IO and DIRECT_MIXED_IO.
FILE_DESCRIPTOR	Specifies a Digital UNIX file descriptor for the Ada file being opened.
	The field value specifies the file descriptor. The file descriptor must be open.

If the file descriptor is not open, if the file descriptor refers to an Ada file that is already open, or if the file descriptor refers to an Ada file with an incompatible mode, the exception USE_ERROR is raised.

The file descriptor option can be used only in the FORM parameter of an OPEN procedure.

PAGE_TERMINATORS | Determines the treatment of line, page, and file terminators. This field applies to text files only.

The field value must be either TRUE or FALSE. A value of TRUE causes the external representation of line, page, and file terminators to be as defined in the DEC Ada run-time reference manuals. A value of FALSE causes the external file to have no page terminators. If no field value is specified, the default value is TRUE.

Output to the external file occurs as follows when the field value is FALSE: line terminators are represented by the character ASCII.LF, page terminators are omitted, and file terminators are represented by the physical end of the file. The exception USE_ERROR is raised when an explicit call is made to the procedure TEXT_IO.NEW_PAGE or an explicit call is made to the procedure TEXT_IO.SET_LINE and the current line number exceeds the value specified by the TO parameter.

Input to the external file occurs as follows when the field value is FALSE: any occurrence of the character ASCII.FF is interpreted as the character ASCII.FF not as a page terminator.

BUFFER_SIZE | Determines the size of the buffer used during file operations.

The field value must be an integer. It specifies the number of bytes in the buffer.

ELEMENT_SIZE | Determines the maximum element size for a direct file.

The field value must be an integer. It specifies the maximum number of bytes in the element.

Each input-output package has an implementation-defined value form string that is used to open or create a file. In general, specification of a FORM parameter is not necessary. The following packages require a value for the FORM parameter to be specified in a CREATE procedure:

- The package DIRECT_IO requires that a maximum element size be specified in the FORM parameter if the item with which the package is instantiated is unconstrained.

- The package DIRECT_MIXED_IO requires that a maximum element size be specified in the FORM parameter.

F.8.2.2 Input-Output Exceptions on Digital UNIX Systems

In addition to the DEC Ada exceptions that apply on all systems, the following also apply on Digital UNIX systems:

- The exception NAME_ERROR is raised as specified in section 14.4: by a call of a CREATE or OPEN procedure if the string given for the NAME parameter does not allow the identification of an external file. On Digital UNIX systems, the value of a NAME parameter can be a string that denotes an Digital UNIX file specification.

  For a CREATE procedure, the value of a NAME parameter can also be a null string in which case it names a temporary external file that is deleted when the main program exits.

- The exception LAYOUT_ERROR is raised as specified in section 14.4: in text input-output by the functions COL, LINE, or PAGE if the value returned exceeds COUNT'LAST. The exception LAYOUT_ERROR is also raised on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, and by attempts to output too many characters to a string.

  In the DEC Ada mixed input-output packages, the exception LAYOUT_ERROR is raised by the procedure GET_ITEM if no more items can be read from the file buffer. It is raised by the procedure PUT_ITEM if the current position exceeds the file buffer size.

F.9 Other Implementation Characteristics

Implementation characteristics relating to the definition of a main program, various numeric ranges, and implementation limits are summarized in the following sections.

F.9.1 Definition of a Main Program

DEC Ada permits a library unit to be used as a main program under the following conditions:

- If it is a procedure with no formal parameters:

  - On OpenVMS systems, the status returned to the OpenVMS environment upon normal completion of the procedure is the value 1.

  - On Digital UNIX systems, the status returned to the Digital UNIX environment upon normal completion of the procedure is the value 0.

  - On Digital UNIX systems, the status returned to the Digital UNIX environment when unhandled exceptions have been raised is the value 42.

- If it is a function with no formal parameters whose returned value is of a discrete type. In this case, the status returned to the operating-system environment upon normal completion of the function is the function value.

- If it is a procedure declared with the pragma EXPORT_VALUED_PROCEDURE, and it has one formal **out** parameter that is of a discrete type. In this case, the status returned to the operating-system environment upon normal completion of the procedure is the value of the first (and only) parameter.

When a main function or a main procedure declared with the pragma EXPORT_VALUED_PROCEDURE returns a discrete value whose size is less than 32 bits (on OpenVMS VAX systems) or 64 bits (on Alpha systems), the value is zero- or sign-extended as appropriate.

F.9.2 Values of Integer Attributes

The ranges of values for integer types declared in the package STANDARD are as follows:

Integer type	Range	Systems on which it applies
SHORT_SHORT_INTEGER	−128 .. 127	All
SHORT_INTEGER	−32768 .. 32767	All
INTEGER	−2147483648 .. 2147483647	All
LONG_INTEGER	−2147483648 .. 2147483647 $−2^{63}$ .. $2^{63}−1$	OpenVMS VAX Alpha

For the applicable input-output packages, the ranges of values for the types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. INTEGER'LAST
POSITIVE_COUNT	1 .. INTEGER'LAST

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. INTEGER'LAST

F.9.3 Values of Floating Point Attributes

DEC Ada provides the following floating point types in the package STANDARD. Additional floating point types are declared in the package SYSTEM (see 13.7b.1 and 13.7b.2).

Type	Default Representation	Systems on which it applies	Section
FLOAT	F_floating IEEE single float	OpenVMS Digital UNIX	3.5.7
LONG_FLOAT	G_floating IEEE double float	OpenVMS Digital UNIX	3.5.7
LONG_LONG_FLOAT	H_floating G_floating IEEE double float	OpenVMS VAX Alpha Digital UNIX	3.5.7

The values of the floating point attributes for the different floating point representations appear in the following tables.

F.9.3.1 F_floating Characteristics

Attribute	F_floating value and approximate decimal equivalent (where applicable)
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON approximately	16#0.1000_000#e−4 9.53674E−07
SMALL approximately	16#0.8000_000#e−21 2.58494E−26
LARGE approximately	16#0.FFFF_F80#e+21 1.93428E+25
SAFE_EMAX	127
SAFE_SMALL approximately	16#0.1000_000#e−31 2.93874E−39
SAFE_LARGE approximately	16#0.7FFF_FC0#e+32 1.70141E+38
FIRST approximately	−16#0.7FFF_FF8#e+32 −1.70141E+38
LAST approximately	16#0.7FFF_FF8#e+32 1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	127
MACHINE_EMIN	−127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.2 D_floating Characteristics

Attribute	D_floating value and approximate decimal equivalent (where applicable)
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0000_000#e−7
approximately	9.3132257461548E−10
SMALL	16#0.8000_0000_0000_000#e−31
approximately	2.3509887016446E−38
LARGE	16#0.FFFF_FFFE_0000_000#e+31
approximately	2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_0000_0000_000#e−31
approximately	2.9387358770557E−39
SAFE_LARGE	16#0.7FFF_FFFF_0000_000#e+32
approximately	1.7014118338124E+38
FIRST	−16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	−1.7014118346047E+38
LAST	16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	1.7014118346047E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	56
MACHINE_EMAX	127
MACHINE_EMIN	−127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.3 G_floating Characteristics

Attribute	G_floating value and approximate decimal equivalent (where applicable)
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON	16#0.4000_0000_0000_00#e−12
approximately	8.881784197001E−16
SMALL	16#0.8000_0000_0000_00#e−51
approximately	1.944692274332E−62
LARGE	16#0.FFFF_FFFF_FFFF_E0#e+51
approximately	2.571100870814E+61
SAFE_EMAX	1023
SAFE_SMALL	16#0.1000_0000_0000_00#e−255
approximately	5.562684646268E−309
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_F0#e+256
approximately	8.988465674312E+307
FIRST	−16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	−8.988465674312E+307
LAST	16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	8.988465674312E+307
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1023
MACHINE_EMIN	−1023
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.4 H_floating Characteristics

Attribute	H_floating value and approximate decimal equivalent (where applicable)
DIGITS	33
MANTISSA	111
EMAX	444
EPSILON approximately	16#0.4000_0000_0000_0000_0000_0000_0000_0#e−27 7.70371977754894341222239117703397E−34
SMALL approximately	16#0.8000_0000_0000_0000_0000_0000_0000_0#e−111 1.10065682146379182109343180020936E−134
LARGE approximately	16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFE_0#e+111 4.54274202684754306593327379930000E+133
SAFE_EMAX	16383
SAFE_SMALL approximately	16#0.1000_0000_0000_0000_0000_0000_0000_0#e−4095 8.40525785778023376565669454330044E−4933
SAFE_LARGE approximately	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_0#e+4096 5.94865747678615882542879663314000E+4931
FIRST approximately	−16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096 −5.94865747678615882542879663314000E+4931
LAST approximately	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096 5.94865747678615882542879663314000E+4931
MACHINE_RADIX	2
MACHINE_MANTISSA	113
MACHINE_EMAX	16383
MACHINE_EMIN	−16383
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.5 IEEE Single Float Characteristics

Attribute	IEEE single float value and approximate decimal equivalent (where applicable)
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#e−4
approximately	9.53674E-07
SMALL	16#0.8000_000#e−21
approximately	2.5849E−26
LARGE	16#0.FFFF_F80#E+21
approximately	1.93428E+25
SAFE_EMAX	125
SAFE_SMALL	
approximately	1.17549E−38
SAFE_LARGE	
approximately	4.25353E+37
FIRST	
approximately	−3.40282E+38
LAST	
approximately	3.40282E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	128
MACHINE_EMIN	−125
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.3.6 IEEE Double Float Characteristics

Attribute	IEEE double float value and approximate decimal equivalent (where applicable)
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON approximately	8.8817841970012E–16
SMALL approximately	1.9446922743316E–62
LARGE approximately	2.5711008708144E+61
SAFE_EMAX	1021
SAFE_SMALL approximately	2.22507385850720E–308
SAFE_LARGE approximately	2.2471164185779E+307
FIRST approximately	–1.7976931348623E+308
LAST approximately	1.7976931348623E+308
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1024
MACHINE_EMIN	–1021
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.4 Attributes of Type DURATION

The values of the significant attributes of the type DURATION are as follows:

DURATION'DELTA	0.0001
DURATION'SMALL	2^{-14}
DURATION'FIRST	–131072.0000

DURATION'LAST   131071.9999
DURATION'LARGE   131071.9999

F.9.5 Implementation Limits

Limit	DEC systems on which it applies	Value
In a subprogram or entry declaration, maximum number of formal parameters that are of an unconstrained record type	All	32
Maximum identifier length (number of characters)	All	255
Maximum number of characters in a source line	All	255
Maximum collection size (number of bytes)	OpenVMS	$2^{31}-1$
	Digital UNIX	$2^{63}-1$
Maximum number of discriminants for a record type	All	245
Maximum number of formal parameters in an entry or subprogram declaration	All	246
Maximum number of dimensions in an array type	All	255
Maximum number of library units and subunits in a compilation closure[1]	All	4095
Maximum number of library units and subunits in an execution closure[2]	All	16383
Maximum number of objects declared with the pragma COMMON_OBJECT or PSECT_OBJECT	All	32757
Maximum number of enumeration literals in an enumeration type definition	All	65535
Maximum number of lines in a source file	All	65534
Maximum number of bits in any object	All	$2^{31} - 1$
Maximum size of the static portion of a stack frame (approximate)	All	2^{30}

[1]The compilation closure of a given unit is the total set of units that the given unit depends on, directly and indirectly.

[2]The execution closure of a given unit is the compilation closure plus all associated secondary units (library bodies and subunits).

G

Ada Language Interpretations

Under the rules of the American National Standards Institute (ANSI), the Ada Joint Program Office (AJPO) of the United States Department of Defense, as the sponsor of the United States Ada language standard, is responsible for the ongoing maintenance and interpretation of that standard.

Under the rules of the International Standards Organization (ISO), ISO-IEC/JTC1/SC22/WG9 (WG9) is responsible for the development and ongoing maintenance and interpretation of the international Ada language standard, coordination with related standards, and so on. Under WG9, an Ada Rapporteur Group (ARG) has been formed to make recommendations on maintenance and interpretation of the standard to WG9.

Both the ANSI (AJPO) and ISO (WG9/ARG) groups coordinate their efforts, and, in particular, seek to retain Ada's strength as an internationally single standard and to avoid any divergence in the interpretation of the ISO and ANSI Ada language standards.

A number of language interpretations (called Ada commentaries) have been made (or recommended) between publication of the standards and the publication of this version of the *DEC Ada Language Reference Manual*. These commentaries have been coordinated between, and are endorsed by, both the ANSI and ISO groups. The summary part of each commentary is presented here for convenience.

For information on how to obtain the full text of the commentaries, contact the AJPO (see the Postscript at the end of this manual for the mailing address), the WG9, or the Ada Information Clearinghouse (AdaIC).

The Ada Information Clearinghouse is operated by the IIT Research Institute for the AJPO. Its post office mailing addresses and phone numbers are as follows:

AdaIC
3D139 (1211 Fern St., C-107)
The Pentagon
Washington, DC 20301-3081
(703) 685-1477

AdaIC
4600 Forbes Blvd.
Lanham, MD 20706
(301) 459-3711

AI-00001/10: Renaming and static expressions [4.9(06), 8.5(04)]

If the name declared by a renaming declaration denotes a constant explicitly declared by a constant declaration having the form specified in 4.9(6), then the name can be used as a primary in a static expression.

AI-00002/07: Deriving homographs for an enumeration literal and a function [8.3(17)]

It is possible to derive both an enumeration literal and a user-defined function that is a homograph of the literal. In such cases, the enumeration literal is hidden by the user-defined function.

AI-00006/05: The subtype of a loop parameter [5.5(06)]

The subtype of a loop parameter is determined by the discrete range in the loop parameter specification. Therefore, the loop parameter has a static subtype if the discrete range is static.

AI-00007/19: Discriminant compatibility for incomplete, private, and access types [3.7.2(05), 3.8.1 (04)]

When a subtype indication with a discriminant constraint is elaborated, 3.3.2(6-8) requires that the compatibility check defined in 3.7.2(5) be performed. The check has two parts: first, check that the value of each discriminant belongs to the corresponding discriminant subtype, and second, if a discriminant is used in a subcomponent constraint, check the constraint for compatibility with the subcomponent's type. If a discriminant constraint is applied to a private type or to an incomplete type before its complete declaration, the second part of the check cannot be performed when the subtype indication is elaborated because no subcomponent declarations exist.

The recommended interpretation of 3.7.2(5) in this case is that the check for subcomponents be deferred and be performed no later than the end of the declaration that allows the deferred check to be completely performed, except when an incomplete type is declared in the private part of a package and its full declaration is given in the package body; in this case, a discriminant constraint is not allowed for the type prior to the end of the package specification.

If a discriminant constraint is given for an access type, the constraint applies to the designated type. The second part of the compatibility check is optional in this case (even if the designated type is completely declared, e.g., even if the constraint occurs in an object declaration or allocator).

When the initial values of discriminants are given by the evaluation of default expressions, the corresponding constraint is checked for compatibility.

AI-00008/05: Negative exponents in based notation [2.4.2(04)]

The exponent of a based literal must not have a minus sign if the based literal is an integer literal.

AI-00014/10: Evaluating default discriminant expressions [3.7.2(08)]

Default discriminant expressions are not evaluated when an explicit initialization expression is provided in an object declaration or a component declaration.

AI-00012/06: Declaration of homographs by derivation and instantiation [8.3(17), 12.3(22)]

Derived subprograms can be homographs, and so can subprograms declared in an instance.

AI-00015/12: When the prefix of ′ADDRESS contains a function name [4.1.4(03), 13.7.2(06)]

The name of an attribute designator can be taken into account when deciding whether the prefix of an attribute is a name or a function call, but the attribute designator cannot be considered when resolving identifiers that are used in the prefix. In particular, the fact that the prefix of ′ADDRESS (as well as the prefix of ′SIZE, ′CONSTRAINED, and ′STORAGE_SIZE) can be an object but not a function call does not affect the resolution of a name that occurs in the prefix.

AI-00016/10: Using a renamed package prefix inside a package [4.1.3(15), 4.1.3(18)]

An expanded name is legal if the prefix denotes a package and the selector is a simple name declared within the visible part of the package, regardless of whether the prefix is a name declared by a renaming declaration.

AI-00018/06: Checking aggregate index and subcomponent values [4.3.2(11)]

The check that the value of an index in an array aggregate belongs to an index subtype can be made before or after all choices have been evaluated. Similarly, the check that a subcomponent value belongs to the subcomponent's subtype can be performed before or after all subcomponent expressions have been evaluated.

AI-00019/07: Checking for too many components in positional aggregates [4.3.2(11)]

CONSTRAINT_ERROR is raised if the bounds of a positional aggregate do not belong to the corresponding index subtype.

AI-00020/07: Real literals with fixed point multiplication and division [4.5.5(10)]

A real literal is not allowed as an operand of a fixed point multiplication or division. The possibility of adopting a more liberal rule in a future version of the language will be studied.

AI-00023/06: Static numeric subtypes [4.9(11), 3.5.4(04), 3.5.7(10), 3.5.9(08)]

Declarations containing integer and real type definitions declare static subtypes, e.g., given

```
type T is range 1 .. 10;
```

T is a static subtype.

AI-00024/09: Type conversions as out parameters for non-scalar types [6.4.1(04)]

If an out parameter has the form of a type conversion and the type mark denotes an array type, the type conversion is performed before the call (see 4.6(11, 13) for the semantics of such a conversion). If the type mark denotes an access type, the value of the variable is converted before the call to the base type of the formal parameter; the designated object need not satisfy a constraint imposed by the formal parameter.

AI-00025/08: Checking out parameter constraints for private types [6.4.1(09)]

When a subprogram parameter has a private type, the constraint checks that are performed before or after the call are those appropriate for the type declared in the private type's full declaration.

AI-00026/07: Effect of full type decl on CONSTRAINED attribute [7.4.2(09)]

After the full declaration of a private type P, P′CONSTRAINED is not allowed unless P's full declaration derives from a private type.

AI-00027/07: Visibility of type mark in explicit conversion or qualified expression [8.3(18)]

The type mark that occurs in an explicit conversion or in a qualified expression must denote a visible declaration.

AI-00030/07: All guards need not be evaluated first [9.7.1(05)]

In the execution of a selective wait statement, the evaluation of conditions, delay expressions, and entry indices is performed in some order that is not defined by the language, except that a delay expression or an entry index cannot be evaluated until after the condition for the corresponding alternative is evaluated and found to be true.

AI-00031/06: Out-of-range argument to pragma PRIORITY [9.8(01)]

If the argument to pragma PRIORITY does not lie in the range of the subtype PRIORITY, the pragma has no effect.

AI-00032/09: Preemptive scheduling is required [9.8(04)]

If an implementation supports more than one priority level, or interrupts, then it must also support a preemptive scheduling policy.

AI-00034/06: Value of COUNT in an accept statement [9.9(06)]

In an accept statement for a member of an entry family, the member being called (and consequently, the task calling the member) is not known until the entry index has been evaluated. This means that if the entry index contains a COUNT attribute, its value is not affected by what member of the family is eventually determined to be called.

AI-00035/06: Body stubs are not allowed in package specifications [10.2(03)]

Body stubs are not allowed in package specifications.

AI-00037/12: Instantiating when discriminants have defaults [12.3.2(04)]

An actual subtype in a generic instantiation can be an unconstrained type with discriminants that have defaults even if an occurrence of the formal type (as an unconstrained subtype indication) is at a place where either a constraint or default discriminants would be required for a type with discriminants.

AI-00038/06: Declarations associated with default names [12.3.6(02)]

The normal visibility rules apply to identifiers used in default subprogram names, i.e., these identifiers are associated with declarations visible at the point of the generic declaration, not those visible at each place of instantiation.

AI-00039/12: Forcing occurrences and premature uses of a type [13.1(06), 13.1(07), 7.4.1(04)]

Each operand of a relational operator (and similarly, the operand of a type conversion or membership test) is considered to be implicitly qualified with the name of the corresponding operand type; such implicit occurrences are considered to be occurrences of the type name with respect to the rules given in 13.1(6), 13.1(7), and 7.4.1(4). This means that such occurrences can be illegal if the implicit type name is an incompletely declared private type (7.4.1(4)),

or they can make the subsequent occurrence of a representation clause illegal (13.1(6, 7)).

AI-00040/07: Multiple specification of T′SIZE, T′STORAGE_SIZE, T′SMALL [13.1(03), 13.6(01)]

For a given type, the ′SIZE, ′STORAGE_SIZE, and ′SMALL attributes can each be specified at most once by an explicit representation clause.

AI-00045/05: Subtype SYSTEM.PRIORITY [13.7(02)]

The subtype SYSTEM.PRIORITY can be non-static. The subtype can have a null range.

AI-00046/06: Lifetime of a temporary file and its name [14.2.1(03), 14.2.1(22)]

1) An implementation is allowed to delete a temporary file immediately after closing it.

2) The NAME function is allowed to raise USE_ERROR if its argument is associated with an external file that has no name, in particular, a temporary file.

AI-00047/08: Effect of RESET on line and page length [14.3.1(04)]

Calling RESET resets the line and page lengths to UNBOUNDED.

AI-00048/12: Default files can be closed, deleted, and re-opened [14.3.2(01), 14.3.1(05)]

The CLOSE operation can be applied to a file object that is also serving as the default input or default output file. The effect is to close the default file. A subsequent OPEN operation can have the effect of opening the default file as well. Similarly, a DELETE operation can be applied to a file object that is serving as the default file.

The exception MODE_ERROR is raised by OPEN if the specified mode is OUT_FILE and the file object being opened is serving as the default input file. Similarly, MODE_ERROR is raised if the specified mode is IN_FILE and the file object being opened is serving as the default output file.

AI-00050/11: When does GET_LINE call SKIP_LINE? [14.3.6(13)]

GET_LINE reads characters until either the end of the string is met or until END_OF_LINE is true. If the end of the string has been met, SKIP_LINE is not called even if END_OF_LINE is true. In particular, no characters are read if the string is null.

AI-00051/07: Reading "integer literals" [14.3.7(06)]

Integer GET reads according to the syntax of an optionally signed numeric literal that does not contain a point. It raises DATA_ERROR if the characters read do not form a legal integer literal. For example, if integer GET attempts to read 0.3, 0E–3, or 20#0#, reading stops before the decimal point for 0.3, after the 3 for 0E–3, and after the second # for 20#0#; DATA_ERROR is raised for 0E–3 since legal integer literals are not allowed to have exponents containing minus signs. DATA_ERROR is also raised for 20#0#, since 20 is not an allowed base value.

AI-00099/12: 'SMALL can be specified for a derived fixed point type [13.2(12)]

A representation clause specifying small for a derived fixed point type is allowed if the resulting model numbers are (representable) values of the parent type and the value specified for small is not greater than the delta of the derived type.

AI-00103/06: Accuracy of a relation between two static universal real operands [4.10(04)]

The relational and membership operations for static universal real operands must be evaluated exactly.

AI-00113/12: A subunit's with clause can name its ancestor library unit [10.5(02)]

A with clause for a subunit can name the subunit's ancestor library name.

AI-00120/05: Overload resolution for assignment statements [8.7(03)]

The type of the right-hand side of an assignment statement can be used to determine an overload resolution of the left-hand side.

AI-00128/04: No membership tests or short-circuit operations in static expressions [4.9(02)]

Membership tests and short-circuit control forms are not allowed in static expressions because neither of these are operators.

AI-00132/05: Static constraints and component clauses [13.4(07)]

A record component clause is only allowed for a record component having a constraint if the constraint is static and, if the component has subcomponents that are constrained, each subcomponent constraint is static.

AI-00137/05: Exponentiation with floating point operand [4.5.6(06)]

Since the model interval for X*X*X*X is sometimes smaller than the interval for (X*X)*(X*X), an implementation cannot compute X**4 as sqr(sqr(X)), where sqr(Y) computes Y*Y. In general, exponentiation to the Nth power must be implemented using $N - 1$ multiplications to ensure the required accuracy is obtained.

AI-00138/10: Representation clauses for derived types [3.4(10), 3.4(22), 13.1(03), 13.6(01)]

If an aspect of a parent type's representation has been specified by an implicit or explicit representation clause and no explicit representation clause is given for the same aspect of the derived type, the representation of the derived and parent types are the same with respect to this aspect.

An explicit length clause for STORAGE_SIZE of a task type, for SIZE (of any type), or for SMALL of a fixed point type, an explicit enumeration representation clause, an explicit record representation clause, or an explicit address clause for a task type can be given for a derived type (prior to a forcing occurrence for the type) even if a representation clause has also been given (explicitly or implicitly) for the same aspect of the parent type's representation. (But only a length clause is allowed for a derived type if the parent type has derivable subprograms.)

An expression in an implicit representation clause is not evaluated when the implicit clause is elaborated.

AI-00139/04: The declaration of "additional operations" for access types [7.4.2(07)]

A consequence of the rule in 7.4.2(7, 8) is that an object A1 can be declared such that the name A1(1) is illegal when A1.all(1) would be legal. Similar examples exist for slices, for the attributes 'FIRST, 'LAST, 'LENGTH, and 'RANGE, and for selection of a record component (e.g., R1.C1 can be illegal when R1.all.C1 is legal).

AI-00143/04: Model numbers for delta 1.0 range −7.0 .. 8.0 [3.5.9(06)]

Given

```
type F is delta 1.0 range -7.0 .. 8.0;
```

The model numbers for F do not include the value 8.0 and F'MANTISSA must be 3.

AI-00144/10: A fixed point type declaration cannot raise an exception [3.5.9(09)]

A fixed point type declaration cannot raise an exception. Declarations such as:

```
type F is delta 2**(-15) range -1.0 .. 1.0;
```

are legal even if F´SIZE is equal to 16 so 1.0 is not a representable value.

AI-00145/04: Dynamic computation of ´MANTISSA for fixed point subtypes [3.5.9(14)]

If F is a non-static fixed point subtype, F´MANTISSA must, in general, be computed at runtime.

AI-00146/10: Model numbers for a fixed point subtype with length clause [3.5.9(14), 3.5.9(16)]

If a length clause specifying small has been given for a fixed point type, T, then the value of small for any subtype of T is given by T´SMALL.

AI-00147/05: Declaring a fixed point type that occupies one word [3.5.9(18)]

A fixed point type that occupies a full word can be declared as:

```
DEL : constant := 1.0/2**(WORD_LENGTH - 1);
type FRACTION is delta DEL range -1.0 .. 1.0 - DEL;
```

AI-00148/05: Legality of −1 .. 10 in loops [3.6.1(02)]

The range −1 .. 10 is illegal as the discrete range in an iteration rule, constrained array type definition, and entry family declaration, since −1 is an expression having a form prohibited by 3.6.1(2), and the other rules of the language do not determine a unique type for the bounds. The possibility of adopting a more liberal rule in a future version of the language will be studied. Note, however, that instead of writing −1 .. 10, one can always write INTEGER range −1 .. 10 or declare −1 as a constant and use the constant name in place of the expression, −1; often it will be appropriate to use the attribute ´RANGE in place of an explicit range such as −1 .. 10.

AI-00149/09: Activating a task before elaboration of its body [3.9(06)]

If an attempt is made to activate a task before its body has been elaborated, PROGRAM_ERROR is raised.

If more than one task is to be activated, the check for unelaborated bodies is performed before an attempt is made to activate any task. Consequently, if PROGRAM_ERROR is raised, no tasks have been activated.

AI-00150/04: Allocated objects belong to the designated subtype [4.8(05)]

CONSTRAINT_ERROR is raised if an object specified by an allocator does not belong to the designated subtype for the allocator.

AI-00151/05: Case expression of a type derived from a generic formal type [5.4(03)]

A type derived from a generic formal type cannot be used in the expression of a case statement.

AI-00153/05: Membership tests cannot use an incompletely declared private type [7.4.1(04)]

The type mark for a private type cannot be used in a membership test before the end of the full declaration of the type. This restriction also applies to the use of a name that denotes a subtype of the private type and the use of a name that denotes any type or subtype that has a subcomponent of the private type.

AI-00154/06: Additional operations for composite and access types [7.4.2(08), 7.4.2(07)]

This Commentary gives details showing which operations are declared for certain composite and access types immediately after their declaration and which are declared later in a package body.

AI-00155/08: Evaluation of an attribute prefix having an undefined value [3.2.1(18), 7.4.3(04)]

The execution of a program is erroneous if the name of a deferred constant is evaluated before the full declaration of the constant has been elaborated. Evaluation of the name of a scalar variable is not erroneous, even if the variable has an undefined value, if the name occurs as the prefix for the attribute ADDRESS, FIRST_BIT, LAST_BIT, POSITION, or SIZE. (In these cases, the value of the variable is not needed.)

AI-00157/05: Overloading resolution and parenthesized expressions [8.7(07)]

The rule requiring aggregates to be given in named notation if they contain a single component association (4.3(4)) is to be considered a syntax rule for purposes of overloading resolution, and in particular, can be used to help resolve the type of a parenthesized expression.

AI-00158/05: The main program is elaborated before it is called [10.5(01)]

The main program is elaborated before it is called.

AI-00163/05: Implicit conversion preserves staticness [4.9(06)]

A static expression is static even if an implicit conversion is applied to it.

AI-00167/05: It is possible to access a task from outside its master [9.4(00)]

A task can be accessed from outside its master, since a function can return a task object as its value, even a task that was activated inside the function.

AI-00169/06: Legality of incomplete null multidimensional array aggregates [4.3(06)]

A multidimensional aggregate is illegal if a value is omitted in the specification of any dimension.

AI-00170/07: Renaming a slice [8.5(05)]

A slice must not be renamed if renaming is prohibited for any of its components.

AI-00172/06: GET_LINE for interactive devices [14.3.6(13)]

An implementation is allowed to assume that certain external files do not contain page terminators. Such external files might be used to represent interactive input devices. (To ensure that such files have no page terminators, an implementation may refuse to recognize any input sequence as a page terminator.) Under such an assumption, GET_LINE and SKIP_LINE can return as soon as a line terminator is read.

AI-00173/05: Completion of execution by exception propagation [9.4(05)]

The execution of a task, block statement, or subprogram is completed if an exception is raised by the elaboration of its declarative part. The execution of a task, block statement, or subprogram is completed if an exception is raised before the first statement following the declarative part and there is no corresponding handler, or if there is one, when it has finished the execution of the corresponding handler. (TASKING_ERROR is the only exception that can be raised under these circumstances. It can be raised by unsuccessful attempts to activate one or more dependent tasks after elaboration of the declarative part and before beginning execution of the sequence of statements.)

AI-00177/04: Use of others in a multidimensional aggregate [4.3.2(08)]

An **others** choice is allowed if an aggregate is not a subaggregate and is the expression of a component association of an enclosing (array or record) aggregate. An **others** choice is also allowed if an aggregate is a subaggregate of a multidimensional array aggregate that is in one of the contexts specified by 4.3.2(5-8).

AI-00179/08: The definition of the attribute FORE [3.5.10(08)]

The attribute ʹFORE is defined in terms of the decimal representation of model numbers.

AI-00180/07: Elaboration checks for INTERFACE subprograms [3.9(05), 13.9(03)]

If a subprogram is named in an INTERFACE pragma, no check need be made that the subprogram body has been elaborated before it is called.

AI-00181/04: NUMERIC_ERROR for nonstatic universal operands [4.10(05)]

When evaluating a nonstatic universal expression, NUMERIC_ERROR can be raised if any operand or the result is a real value that lies outside the range of safe numbers of the most accurate predefined floating point type (excluding *universal_real*) or an integer value that lies outside the range SYSTEM.MIN_ INT .. SYSTEM.MAX_INT.

AI-00186/08: Pragmas recognized by an impl do not force default representation [13.1(06), 2.8(09)]

The intent of 2.8(9) is that an invalid pragma have the same effect as if it were absent. To ensure that this intent is realized, a pragma defined by the Standard or by an implementation is not allowed to contain an occurrence of a name or expression that forces the determination of the default representation of a type, since such occurrences would make later representation clauses for the type illegal. Consequently, a representation clause for a type can be accepted even if the clause is given after a pragma that contains an expression that normally would force the default representation of the type to be determined (since such a pragma will be considered invalid, and ignored). (See AI-00322 for a similar rule for pragmas whose identifiers are not defined either by the Standard or by the implementation.)

AI-00187/06: Using a name decl by a renaming decl as an expanded name selector [4.1.3(15)]

A name declared by a renaming declaration can be used as a selector in an expanded name.

AI-00190/06: A static expression cannot have a generic formal type [4.9(02), 4.3.2(03)]

A static expression is not allowed to have a generic formal type (including a type derived from a generic formal type, directly or indirectly). (Consequently, if an array's index subtype is a generic formal type, aggregates for that dimension of the array can have only a single component association and this component association must have a single choice.)

AI-00192/05: Allowed names of library units [8.6(02)]

The name of a library unit cannot be a homograph of a name that is already declared in package STANDARD.

AI-00193/05: The value of 'FIRST's argument in overloading resolution [8.7(08)]

Any overloaded identifiers occurring in the argument for 'FIRST(N), 'LAST(N), and 'RANGE(N) must be resolved independently of the context in which these attributes are used.

AI-00195/09: The intended use of CLOCK [9.6(05)]

CLOCK returns a value that reflects the time of day in the external environment.

AI-00196/05: Use of 86_400.0 in TIME_OF [9.6(06)]

If TIME_OF is called with a seconds value of 86_400.0, the value returned is equal to TIME_OF for the next day with a seconds value of 0.0. In addition, the SECONDS function always returns a value less than 86_400.0, even if the SECONDS argument of TIME_OF was 86_400.0.

AI-00197/07: With SYSTEM clause not needed for pragma PRIORITY [9.8(00)]

Use of the pragma PRIORITY does not require the package SYSTEM to be named in a with clause for the enclosing compilation unit.

AI-00198/09: Termination of unactivated tasks [9.3(04), 9.10(05), 9.3(08)]

If a task is abnormally completed, then any task it has created but not yet activated becomes terminated and is never activated.

If PROGRAM_ERROR is raised before attempting to activate one or more tasks because the body of at least one of these tasks has not yet been elaborated (see AI-00149), all the unactivated tasks become terminated.

AI-00199/08: Implicit declaration of library subprograms [10.1(06)]

A subprogram body given in a compilation unit (following the context clause) is interpreted as a secondary unit if the program library already contains a subprogram declaration or generic subprogram declaration having the same identifier as the body. Otherwise, the subprogram body is interpreted as a library unit.

Successfully compiling a subprogram body that is a library unit means the unit is added to the library, replacing any previously existing library unit having the same identifier. (The previously existing library unit can only be a package declaration, a generic package declaration, a generic instantiation, or a previously compiled subprogram body that is a library unit.)

AI-00200/08: Dependences created by inline of generic instantiations [10.3(07), 6.3.2(03)]

If inline inclusion of a subprogram call is achieved due to pragma INLINE, an implementation is allowed to create a dependence of the calling unit on the subprogram body; when such a dependence exists, the unit containing the call is obsolete if the subprogram body is obsolete. Such dependences can be created even when the subprogram is created as a result of a generic instantiation.

AI-00201/07: The relation between TICK, CLOCK, and the delay statement [9.6(01), 9.6(05), 9.6(04), 13.7.1(07)]

The value returned by successive calls to the CLOCK function can be expected to change at the frequency indicated by SYSTEM.TICK.

There is no required relation between SYSTEM.TICK and DURATION'SMALL.

Delay statements need not be executed with an accuracy that is related to SYSTEM.TICK or DURATION'SMALL; in particular, delay statements can be executed more accurately than SYSTEM.TICK implies. Execution with less accuracy than SYSTEM.TICK requires justification in terms of AI-00325.

AI-00205/06: The formula for MANTISSA is correct [3.5.7(06)]

The number of mantissa bits for D decimal digits of accuracy is correctly given by the formula in the Standard, namely, the integer next above $(D*\log(10)/\log(2)) + 1$.

AI-00209/06: Exact evaluation of static universal real expressions [4.10(04)]

An implementation can refuse to evaluate a static universal real expression only if there are insufficient resources to evaluate the expression exactly, e.g., if there is insufficient memory available. Inexact results must not be delivered.

AI-00210/04: Loop name in an exit statement as an expanded name [5.7(02)]

An expanded name is allowed as the loop name in an exit statement.

AI-00215/05: Type of EXP should be FIELD [14.3.8(20)]

The type of the EXP parameter for PUT is FIELD, not INTEGER.

AI-00217/05: The safe numbers of a floating point subtype [3.5.7(09)]

The safe numbers of a floating point subtype are the safe numbers of its base type.

AI-00219/06: Use of & and 'IMAGE in static expressions [4.9(02)]

In a static expression, every factor, term, simple expression, and relation must have a scalar type.

AI-00225/09: Secondary units for generic subprograms [10.1(06)]

If a subprogram body given in a compilation unit has the same identifier as a library unit and the library unit is a generic subprogram declaration, then the subprogram body is interpreted as a secondary unit.

AI-00226/06: Applicability of context clauses to subunits [10.1.1(04)]

The context clause of a library unit that is a declaration applies not only to the secondary unit that defines the corresponding body, but also to any subunits of the secondary unit.

AI-00231/03: Full declarations of incomplete types can have discriminants [3.8.1(04)]

The full declaration of an incomplete type can be a derived type with unconstrained discriminants when no discriminant part is given in the incomplete type's declaration.

AI-00232/05: Full declarations that implicitly declare unconstrained types [7.4.1(03)]

A full declaration of a private type can declare a constrained array subtype or a constrained type with discriminants.

AI-00234/05: Lower bound for 'IMAGE of enumeration values [3.5.5(10)]

The lower bound of the string returned by the predefined attribute IMAGE is one.

AI-00235/05: Redundant parentheses enclosing universal_fixed expressions [4.5.5(11)]

An expression having type *universal_fixed* can be enclosed in parentheses before being converted to some other numeric type.

AI-00236/12: Pragma ELABORATE for bodiless packages with tasks [10.5(04)]

If a package declares a task object but no package body is required or provided by a programmer, 9.3(5) says an implicit body is provided. The effect of a pragma ELABORATE that names such a package is to require that the implicitly provided package body be elaborated, thereby activating the task declared in the package.

AI-00237/06: Instances having implicit package bodies [12.3(17), 9.3(05)]

Given a generic package declaration that does not require a body and that has no explicit body, when the generic package is instantiated, if the instance specification requires a body, then an implicit instance body is created and is elaborated when the instantiation is elaborated.

AI-00239/11: ENUMERATION_IO and IMAGE for non-graphic characters [14.3.9(06), 14.3.9(09), 3.5.5(11)]

If ENUM_IO is an instantiation of ENUMERATION_IO for a character type that contains a non-graphic character, e.g.,

```
package ENUM_IO is new ENUMERATION_IO (CHARACTER);
```

then for each non-graphic character (such as ASCII.NUL), ENUM_IO.PUT should output the corresponding sequence of characters used in the type definition (e.g., PUT(ASCII.NUL) should output the string "NUL" if SET has the value UPPER_CASE and WIDTH is less than 4). Furthermore, ENUM_IO.GET should be able to read the sequence of characters output by ENUM_IO.PUT for a non-graphic character, returning in its ITEM parameter the corresponding enumeration value.

Similarly, the image of a non-graphic character (i.e., the result returned for the attribute designator IMAGE) should be the sequence of characters used in the type definition of CHARACTER (e.g., CHARACTER'IMAGE(ASCII.NUL) = "NUL"),and 'VALUE should accept such a string as representing the corresponding enumeration value.

An implementation conforms to the Standard in this respect if the result produced by 'IMAGE for a non-graphic character is accepted by 'VALUE, and if the result (if any) produced by PUT can be read by GET; GET is also allowed to raise DATA_ERROR when attempting to read any string produced by PUT for a non-graphic character.

This interpretation is non-binding, i.e., implementers are encouraged to conform to it but are not required to do so by the validation tests. A future version of the Standard may incorporate this interpretation.

AI-00240/05: Integer type definitions cannot contain a RANGE attribute [3.5.4(03)]

A range attribute is not allowed in an integer type definition.

AI-00242/09: Subprogram names allowed in pragma INLINE [6.3.2(03), 2.8(09)]

If the pragma INLINE appears at the place of a declarative item, every name in the pragma must denote at least one subprogram or generic subprogram declared explicitly earlier in the same declarative part or package specification. If the pragma appears after a given library unit, the pragma must contain just the name of the library unit, and the library unit must be a subprogram or a generic subprogram. If a pragma INLINE appears at the place of a declarative item and a name in the pragma is overloaded, the pragma applies just to those subprograms whose declarations occur (explicitly) earlier in the same declarative part or package specification. If a name in a pragma INLINE is declared by a renaming declaration, and the denoted subprogram is explicitly declared earlier in the same declarative part or package specification, inline expansion is desired for every call of the denoted subprogram (whether the call uses the new or the old name). If a pragma INLINE applies to a subprogram, inline expansion is desired for every call of the subprogram, whether or not the call uses a name declared by a renaming declaration.

AI-00243/05: Overriding width format in TEXT_IO [14.3.5(07)]

If the specification of WIDTH or FORE in a call of PUT is insufficiently large, the output is given with no leading spaces.

AI-00244/04: Record aggregates with multiple choices in a component association [4.3.1(01)]

In a record aggregate, a component association having multiple choices denoting components of the same type is considered equivalent to a sequence of single choice component associations representing the same components.

AI-00245/08: Type conversion conformance for renamed subprogram/entry calls [6.4.1(03), 8.5(08)]

When a type conversion is used as an actual parameter corresponding to an **in out** or **out** formal parameter and the subprogram being called was declared by a renaming declaration (renaming either a subprogram or entry), the name given as the type mark (in the type conversion) must conform to the name given for the corresponding parameter of the denoted subprogram or entry (not the name given in the renaming declaration).

AI-00247/05: A non-null FORM argument can be required by an implementation [14.2.1(03)]

An implementation can require that a non-null FORM argument be given to CREATE and/or OPEN by raising USE_ERROR if one is not provided.

AI-00251/05: Are types derived from generic formal types static subtypes? [4.9(11)]

Types derived from generic formal types are not static subtypes.

AI-00257/04: Restricting generic unit bodies to compilations [10.3(09)]

10.3(9) allows an implementation to require that bodies and subunits of a generic unit appear in the same compilation. An implementation is allowed to apply this rule selectively, i.e., the conditions under which an implementation requires placement in a single compilation may depend on characteristics of the generic specification, body, or subunit.

AI-00258/06: ′POSITION etc. for renamed components [13.7.2(07), A(34)]

The prefix for ′POSITION, ′FIRST_BIT, and ′LAST_BIT must have the form R.C, where R is a name denoting a record and C is the name of a component of the record.

AI-00260/06: Limited "full types" [7.4.4(04)]

A full type declaration declares a limited type if an assignment operation is not visible for the type of some subcomponent at the place of the full type declaration.

A formal parameter whose type mark denotes an incompletely declared private type cannot have mode OUT if the parameter's full type declaration declares a limited type.

AI-00261/03: "any error" –> "any illegal construct" [10.3(03)]

An attempt to compile an illegal compilation unit has no effect on the program library (see also AI-00255).

AI-00263/06: A named number is not an object [3.2(08), 13.5(04), 13.7.3(03), 13.7.3(05)]

A number declaration declares a named number, which is not an object.

The elaboration of a number declaration proceeds by evaluating the initialization expression and creating a named number. The value of the initialization expression then becomes the value of the named number.

AI-00265/05: Index subtype of an array aggregate [4.3.2(11)]

The index subtype of an array type declared by a constrained array definition is the subtype defined by the corresponding discrete range.

AI-00266/09: A body cannot be compiled for a library unit instantiation [10.1(06)]

If a generic package is instantiated as a library unit, it is illegal to attempt to compile a package body having the same identifier as that of the instantiation.

After instantiating a generic subprogram as a library unit, any attempt to compile a subprogram body having the same identifier as that of the library unit instantiation causes the instantiation to be deleted from the library and replaced with the new library unit subprogram.

AI-00267/06: Evaluating expressions in case statements [3.5.4(10), 5.4(06), 11.6(06)]

An exception is raised if the value of the expression in a case statement does not belong to the base type of the expression.

AI-00268/06: Activation of already abnormal tasks [9.3(03)]

If a task is aborted before it is activated, no exception is raised when an attempt is made to activate the task.

AI-00276/07: Rendezvous that are "immediately possible" vs. timed entry calls [9.7.3(04), 9.7.2(01)]

A timed entry call with a zero or negative delay issues an entry call that is canceled only if a rendezvous is not immediately possible. 9.7.2(4) specifies the conditions under which an entry call is immediately possible. In a distributed implementation of Ada, it may take a non-negligible amount of time to determine whether an entry call is "immediately" possible.

AI-00279/09: Exceptions raised by calls of I/O subprograms [14.1(11)]

Any exception can be raised when evaluating the actual parameters of a call of an input-output subprogram. In addition, STORAGE_ERROR can be raised by the call itself before execution of the body has begun. But once execution of the body of an input-output subprogram has been started, the only exceptions that can be propagated to the caller are the exceptions defined in the package IO_EXCEPTIONS and the exceptions PROGRAM_ERROR and STORAGE_ERROR. In particular, if CONSTRAINT_ERROR, NUMERIC_ERROR, or TASKING_ERROR is not raised by the evaluation of any argument, then none of these exceptions will be raised by the call. Furthermore, PROGRAM_ERROR can only be raised due to errors made by the user of the input-output subprogram.

AI-00282/06: Compatibility of constraint defined by discrete range [3.6.1(04)]

The constraint defined by a discrete range is compatible with a subtype if each bound of the discrete range belongs to the subtype, or if the discrete range defines a null range; otherwise the constraint is not compatible with the subtype.

AI-00286/11: Declarations visible in a generic subprogram decl and body [12.1(05), 8.4(05), 8.3(15)]

Except within the body of a generic subprogram, the declaration of a generic unit is not a declaration for which overloading is allowed. In particular, any declarations occurring in an outer declarative region or made potentially visible by a use clause are not directly visible in the generic formal part if they have the same identifier as the subprogram.

Within the body of a generic subprogram, overloading is defined for the generic subprogram declaration in the same way as for a nongeneric subprogram. In particular, overloadable declarations occurring in an outer declarative region or made potentially visible by a use clause can be directly visible in the body even though they are not directly visible in the generic formal part.

AI-00287/05: Resolving overloaded entry calls [9.5(05), 8.7(13)]

A call to an overloaded entry is resolved using the same kind of information as is used for resolving overloaded procedure calls (the name of the entry, the number of parameters, the types and the order of the actual parameters, and the names of the formal parameters).

AI-00288/06: Effect of priorities during activation [9.8(04)]

A task activation should be performed with the priority of the task being activated or the priority of the task causing the activation, whichever is higher.

AI-00289/05: Ancestor unit names in separate clauses must be simple names [10.2(05)]

If P is a library unit, then the name in a "separate" clause for a subunit of P must be P and not STANDARD.P.

AI-00292/05: Derived types with address clauses for entries [3.4(10), 13.5(08)]

An address clause applied to an entry of a task type also applies to a type derived (directly or indirectly) from the task type.

AI-00293/05: Null others choice for array aggregates [4.3(06)]

The others choice in an array aggregate can specify no components.

AI-00294/05: The name given in pragma CONTROLLED [4.8(11)]

The type name given in a pragma CONTROLLED cannot be declared by a subtype declaration nor can it be a first named subtype since a derived type is not allowed.

AI-00295/05: Evaluating the variable in an actual parameter type conversion [6.4.1(04)]

For an actual parameter (of any type) of mode **in out** or **out** that is a type conversion, the variable name is evaluated before the call and therefore determines the denoted entity.

AI-00298/05: Interaction between pragmas ELABORATE and INTERFACE [10.5(04), 13.9(03)]

A pragma ELABORATE can be applied to a library unit whose body is supplied by a pragma INTERFACE.

AI-00300/07: Prefixes of attributes in length clauses [13.2(02)]

The prefix of the attribute that appears in a length clause must be a simple name. An expanded name, or the name T'BASE, is not allowed.

AI-00305/05: T'ADDRESS when T is a task type yields the task object address [13.7.2(03)]

If T denotes a task type, then within the body of task unit T, the T in T'ADDRESS is considered to refer to the name of the task object that designates the task currently executing the body, i.e., T'ADDRESS returns the address of the object.

AI-00306/15: Pragma INTERFACE; allowed names and illegalities [2.8(09), 13.9(03)]

If a pragma INTERFACE names a language that is acceptable to an implementation, the subprogram name must denote one or more subprograms declared explicitly earlier in the same declarative part or package specification. (The pragma has no effect if no named subprogram satisfies the requirements.) The pragma is applied to all such subprograms other than enumeration literals and subprograms declared by generic instantiation.

If a subprogram named in the pragma was declared by a renaming declaration, the pragma applies to the denoted subprogram, but only if the denoted subprogram otherwise satisfies the above requirements.

It is illegal to apply a pragma INTERFACE to a subprogram for which a pragma INTERFACE has already been applied.

If a pragma INTERFACE applies to a subprogram, it is illegal to provide a body for the subprogram.

AI-00307/04: GET at end of file and from a null string [14.3.7(06), 14.3.7(14), 14.3.8(09), 14.3.8(18), 14.3.9(06), 14.3.9(11)]

END_ERROR is raised (not DATA_ERROR) when attempting to read integer, real, or enumeration values from a string that is null or that only contains blanks. END_ERROR (not DATA_ERROR) is raised when attempting to read integer, real, or enumeration values from a file that has no remaining elements or whose only remaining elements are blanks, line terminators, or page terminators.

AI-00308/05: Checking default initialization of discriminants for compatibility [3.2.1(16)]

When an object of a type with discriminants is created either by an object declaration or an allocator, and the values of the object's discriminants are determined by default, each discriminant value is checked for compatibility, as defined in 3.7.2(5). CONSTRAINT_ERROR is raised if this check fails.

AI-00310/04: OTHERS choices and static index constraints [4.3.2(03)]

An others choice is static if the corresponding index subtype is static and if the corresponding index bounds were specified with a static discrete range in the applicable index constraint.

AI-00311/06: No NUMERIC_ERROR for null strings [11.1(06)]

When computing the upper bound of a null string literal, NUMERIC_ERROR must not be raised, even if the lower bound has no predecessor (but see AI-00325).

AI-00312/04: NUMERIC_ERROR when evaluating null aggregates and slices [11.1(06)]

When determining the length of a null aggregate or slice, it is usually easy for an implementation to avoid raising NUMERIC_ERROR. This exception should be raised in these circumstances only when relevant restrictions in the execution or compilation environment make it impractical or impossible to avoid raising the exception (see AI-00325).

AI-00313/03: Non-null bounds belong to the index subtype [4.3.2(11), 4.6(13)]

CONSTRAINT_ERROR is raised if a non-null choice of an aggregate does not belong to the corresponding index subtype.

For conversion to an unconstrained array type, CONSTRAINT_ERROR is raised if a non-null dimension of the operand has bounds that do not belong to the corresponding index subtype of the target type.

AI-00314/05: The safe numbers for IBM-370 floating point [3.5.7(09)]

The safe and model numbers for IBM-370 32-bit floating point have the following characteristics:

```
DIGITS=6
MANTISSA = 21
EMAX = 84
SAFE_EMAX = 252
```

AI-00316/05: Definition of blank, inclusion of horizontal tab [14.3.9(06)]

A blank is defined as a space or a horizontal tabulation character.

AI-00319/09: Checking for subtype incompatibility [3.7.2(05), 3.8.1(04)]

No object can have a subcomponent with an incompatible discriminant or index constraint. In particular, even when a discriminant constraint is applied to a private type before its full declaration or to an incomplete type (before its full declaration) and a discriminant is used to constrain a subcomponent, no object of the type can be created if it would have a subcomponent with an incompatible discriminant or index constraint.

AI-00320/06: Sharing external files [14.3.5(03), 14.2.4(00), 14.3.4(00), 14.2.2(00)]

If several file objects are associated with the same external file, some effects are implementation dependent. For example, if two sequential file objects are associated with the same external file, applying a read or write operation to one file object can change the effect of applying these operations (or the end of file operation) to the other file object. Other effects are specified by the language. In particular, if two text file objects are associated with a single external file (e.g., a terminal), the page, line, and column numbers for the output file object cannot be updated implicitly after reading from the input file object, and vice versa.

AI-00321/02: Forcing occurrence of index subtype [13.1(06)]

A forcing occurrence of the name of an array type or subtype forces the default determination of each index subtype, and similarly, for forcing occurrences of any type or subtype having a subcomponent of such an array type.

AI-00322/02: Forcing occurrences in unknown pragmas [13.1(06), 2.8(09)]

An occurrence of a name within an expression is not a forcing occurrence if the expression occurs in a pragma whose identifier is not defined either by the Standard or by the implementation.

AI-00324/06: Checking the subtype of a non-null access value [3.8(06)]

An access value of type T belongs to every subtype of T if T's designated type is neither an array type nor a type with discriminants.

AI-00325/05: Implementation-dependent limitations [1.1.2(00)]

Implementation-dependent limitations must be justified. An implementation-dependent limitation is justified if it is impossible or impractical to remove it, given an implementation's execution environment.

AI-00328/08: Legality of uninstantiated generic units [12.2(01)]

The legality of a generic unit must be checked even if the generic unit is never instantiated.

AI-00330/12: Explicit declaration of enumeration literals [8.3(17), 3.5.1(03), 3.3.3(01)]

If an enumeration literal is declared with an enumeration type definition, then a function having the same identifier as the enumeration literal and the same parameter and result type profile cannot also be declared immediately within the same declarative region. Similarly, a non-overloadable declaration of the enumeration literal's identifier is not allowed immediately within the declarative region containing the enumeration type definition.

AI-00331/07: The effect of a constraint in an allocator [4.8(05)]

When a discriminant or index constraint is imposed on the type mark in an allocator and the type mark denotes an access type, the constraint does not affect the subtype of the allocated object (which in this case has an access value).

Similarly, when the type mark in an allocator denotes a scalar type, the subtype denoted by the type mark does not affect the subtype of the allocated (scalar) object.

AI-00332/04: NAME_ERROR or USE_ERROR raised when I/O not supported [14.2.1(04), 14.2.1(07), 14.4(04), 14.4(05)]

CREATE and OPEN can raise USE_ERROR or NAME_ERROR if file creation or opening is not allowed for any file.

AI-00336/05: Address clause for subprogram bodies [13.5(05)]

An address clause cannot be given for a subprogram whose body acts as its declaration.

AI-00339/04: Allow non-English characters in comments [2.7(01), 2.1(01)]

An implementation is allowed (but not required) to accept an extended character set (i.e., graphic characters whose codes do not belong to the ISO seven-bit coded character set (ISO standard 646)) as long as the additional characters appear only in comments.

AI-00343/05: Decimal fixed point representations [3.5.9(10)]

An implementation can use decimal or binary representations for fixed point values as long as all model numbers are represented exactly.

AI-00350/04: Lexical elements not changed by allowable character replacements [6.3.1(05), 2.10(05)]

Lexical elements differing only in their use of allowable replacements of characters (as defined in 2.10) are considered as the same. In particular, use of the allowable replacements does not affect the conformance of formal parts, discriminant parts, or actual parameters.

AI-00354/03: On the elaboration of library units [10.5(02)]

There is no requirement that the body of a library unit be elaborated as soon as possible after the library unit is elaborated. In particular, the pragma ELABORATE should be used if it is important that a library package's body be elaborated before another package is elaborated.

AI-00355/06: Pragma ELABORATE for predefined library packages [10.5(04), 9.6(07), 13.7(02), 13.10.1(01), 13.10.2(01), 14.1(01), 14.3(01), 14.6(05), C(22)]

An attempt to use an entity declared within a predefined library unit or a unit declared within a predefined library package must raise PROGRAM_ERROR if a required body has not been elaborated (3.9(5-8)). An implementation is not allowed to raise PROGRAM_ERROR for this reason, however, if a pragma ELABORATE has been given for the library unit. (In particular, if the library unit body provided by the implementation depends on other (implementation-defined) library units, the implementation must ensure prior elaboration of the required bodies, e.g., by providing appropriate ELABORATE pragmas.)

AI-00356/08: Access values that designate deallocated objects [3.2.1(18), 4.8(07), 13.10.1(06)]

The storage occupied by a designated object can be reclaimed immediately after applying an instance of the unchecked deallocation procedure to an access variable that designates the object.

If two objects having non-null access values designate the same object and an instance of the unchecked deallocation procedure is applied to one of the objects, the other object is considered to have an undefined value; any attempt to use such a value makes execution of the program erroneous.

Similarly, if a name declared by a renaming declaration denotes a subcomponent of an object that is later freed by calling an instance of the unchecked deallocation procedure, the name is considered to have an undefined value; any attempt to evaluate the name (e.g., by assigning a value to it) makes execution of the program erroneous.

AI-00357/05: CLOSE or RESET of a sequential file from OUT_FILE mode [14.2.1(09), 14.2.1(15)]

If a sequential input-output file having mode OUT_FILE is closed or reset, the most recently written element since the last open or reset is the last element that can be read from the file. If no elements have been written, the closed or reset file is empty. (As a consequence, opening a sequential input-output file with mode OUT_FILE or resetting a sequential input-output file to mode OUT_FILE has the effect of deleting the previous contents of the file.)

AI-00358/10: Discriminant checks for non-existent subcomponents [3.7.2(05), 3.7(08)]

When checking the compatibility of a discriminant constraint, 3.7.2(5) requires that a discriminant's value be substituted in component subtype definitions that depend on the discriminant. This substitution is performed only for those subcomponents that exist in the subtype defined by the constraint.

AI-00362/03: "component of a record" for representation attributes [13.7.2(08)]

The prefix of 'POSITION, 'FIRST_BIT, or 'LAST_BIT must denote a component of a record object.

AI-00365/05: Actual parameter names are evaluated in generic instantiations [12.3(17)]

In a generic instantiation, the names appearing as actual parameters are evaluated.

AI-00366/07: The value of SYSTEM.TICK for different execution environments [13.7.1(07)]

SYSTEM.TICK should have a value that reflects the precision of the clock in the main program's execution environment. If SYSTEM.TICK does not have an appropriate value, the effect of executing the program is not defined.

AI-00367/06: Deriving from types declared in a generic package [3.4(11), 12.1(05)]

The rules concerning derivable subprograms in the visible part of a nongeneric package are applicable in the visible part of a generic package. (The effect of a derived type declaration in an instance of a generic unit is discussed in AI-00398.)

AI-00370/06: Visibility of subprogam names within instantiations [8.3(16)]

Any declaration with the same designator as a subprogram instantiation is not visible, even by selection, within the instantiation.

AI-00371/05: Representation clauses containing forcing occurrences [13.1(07), 2.8(09)]

An expression in a representation clause is illegal if it contains a forcing occurrence for the type whose representation is being specified.

AI-00374/06: An attempt to access an undefined constant is erroneous [3.2.1(18)]

The execution of a program is erroneous if it attempts to evaluate a scalar constant with an undefined value.

AI-00375/05: Restricting the allowed values of a floating point subtype [3.5.7(17)]

If a floating point constraint in a subtype indication includes a range constraint, the range of the values that belong to the subtype (i.e., that satisfy the constraint) is defined by the range constraint. If no range constraint is present, the range of values that belong to the subtype is not affected, even though the accuracy of the subtype may be reduced.

AI-00376/04: Universal real operands with fixed point * and / [4.5.5(10)]

An expression having type *universal_real* is not allowed as an operand of a fixed point multiplication or division operation. The possibility of adopting a more liberal rule in a future version of the language will be studied.

This commentary extends the conclusions of AI-00020 to cover all expressions of type *universal_real*, not just those having the form of a real literal.

AI-00379/03: Address clauses for entries of task types [13.5(08)]

If an interrupt is linked to an entry of more than one task object (of the same type), the program is erroneous.

AI-00384/05: Use of an incomplete private type in a formal type declaration [7.4.1(04)]

An incompletely declared private type cannot be used in the declaration of a generic formal type.

AI-00387/05: Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR [11.1(06), 3.5.4(10), 3.5.6(06), 4.5(07), 4.5.5(12), 4.5.7(07), 4.10(05)]

Wherever the Standard requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR should be raised instead. This interpretation is non-binding.

AI-00388/06: Pragmas are allowed in a generic formal part [2.8(04)]

Pragmas are allowed in a generic formal part.

AI-00396/03: Correction to discussion of AI-00025 [6.4.1(09)]

The discussion section of AI-00025/07 should be corrected. Instead of saying,

> "The effect of the call, CALL_Q_NOW.CALL_Q.Q(Y), is to assign an invalid value to P.Z"

the discussion should say,

> "The effect of elaborating CALL_Q_NOW is to assign an invalid value to P.Z".

AI-00397/04: Checking the designated subtype for an allocator [4.8(13), 4.8(05)]

When evaluating an allocator, a check is made that the designated object belongs to the allocator's designated subtype. CONSTRAINT_ERROR is raised if this check fails. This check can be made any time before evaluation of the allocator is complete. In particular, it is not defined whether this check is performed before creation of a designated object, evaluation of any default initialization expressions, or evaluation of any expressions contained in the allocator.

AI-00398/08: Operations declared for types declared in instances [12.3(05), 3.4(11)]

If the parent type in a derived type definition is a generic formal type, the operations declared for the derived type in the template are determined by the class of the formal type. The operations declared for the derived type in the instance are determined by the type denoted by the formal parameter.

Similarly, if the component type of an array type is a generic formal type or if the designated type of an access type is a generic formal type, the operations declared for the array and access type in the template depend on the class of the formal type. If the array and access type declarations do not occur in the generic formal part, then the operations declared for these types in a generic instance are determined by the type denoted by the formal parameter in the instance.

If the designated type in an access type declaration is an incomplete type, additional operations can be declared for the access type by the full declaration of the incomplete type (7.4.2(7-8)). If the full declaration declares a type derived from a generic formal type, the additional operations (if any) declared for the access type in the template are determined by the class of the formal type. The additional operations declared for the access type in the instance are determined by the type denoted by the formal parameter.

Similar rules apply when the parent type, component type, or designated type is derived, directly or indirectly, from a generic formal type.

AI-00405/06: One nonstatic operand for a universal real relation [4.10(04)]

If the operands of a relational operator or membership test have the type *universal_real* and one or more of the operands is nonstatic, the static operands must be evaluated exactly. Doing so, however, does not impose a run-time overhead.

AI-00406/05: Evaluating parameters of a call before raising PROGRAM_ERROR [3.9(05)]

It is not defined whether the check that the body of a subprogram has been elaborated is made before or after the actual parameters of a call have been evaluated.

Similarly, it is not defined whether the check that the body of a generic unit has been elaborated is made before or after the generic actual parameters of an instantiation have been evaluated.

AI-00407/06: The operations of a subtype with reduced accuracy [3.5.8(16), 3.5.10(15), 4.5.7(00), 5.2(03)]

When assigning a fixed or floating point value to a variable, the stored value need only be represented as a model number of the variable's subtype. Furthermore, if no exception is raised by the assignment, the stored value belongs to the subtype of the variable.

If a real subtype is used as the type mark in a membership test, qualification, or explicit conversion, the corresponding operation is performed with the accuracy of the base type and the range of the subtype.

For a real subtype, the value of the attributes FIRST or LAST is represented with at least the accuracy of the base type. The values of other attributes of a real subtype are given exactly.

AI-00408/11: Effect of compiling generic unit bodies separately [10.3(06)]

An implementation is allowed to create a dependence on a generic unit body such that successfully compiling (or recompiling) the body separately makes previously compiled units obsolete if they contain an instantiation of the generic unit. A similar dependence can be created for separately compiled subunits of a generic unit.

AI-00409/05: Static subtype names created by instantiation [12.3(05), 4.9(11)]

A subtype can be nonstatic in a generic template and static in a corresponding instance.

AI-00412/06: Expanded names for generic formal parameters [4.1.3(15), 4.1.3(18), 12.1(05)]

A formal parameter of a generic unit can be denoted by an expanded name.

AI-00418/06: Self-referencing with clauses [10.1(03)]

Circular dependences among library units are not allowed, i.e., the library unit being compiled cannot have the same name as a previously compiled library unit if a with clause for the unit being compiled establishes a direct or indirect dependence on the previously compiled unit.

AI-00422/06: Representation clauses for derived enumeration and record types [13.1(03), 13.3(02), 13.4(02)]

An enumeration representation clause or a record representation clause can be given for an enumeration type or a record type declared by a derived type declaration.

The index subtype for the aggregate used in an enumeration representation clause is the base type of the enumeration type.

A record representation clause for a first named subtype can specify the representation of any component that belongs to the record's base type, even if the subtype is constrained.

AI-00425/05: Restrictions on arguments of implementation-defined pragmas [2.8(08)]

An implementation can impose restrictions on the arguments of its implementation-defined pragmas (e.g., it can require that an argument be a static expression).

AI-00426/05: Operations on undefined array values [3.2.1(18), 4.5.1(03)]

If both operands of a predefined logical operator do not have the same number of components, CONSTRAINT_ERROR is raised, even if one of the operands has a scalar component with an undefined value.

AI-00430/05: Using an enumeration literal does not raise PROGRAM_ERROR [3.5.1(03), 3.9(08)]

The use of an enumeration literal (i.e., a call of the corresponding parameter-less function) does not raise PROGRAM_ERROR.

AI-00431/05: Boolean operators producing out of range results [4.5.1(03)]

Predefined logical operations on boolean arrays are performed on a component-by-component basis, using the predefined logical operation for the component type (even if a user-defined logical operation for the component type is visible and hides the predefined one).

AI-00441/06: A task without dependents can be completed but not terminated [9.4(06)]

A task that has no dependent tasks can be completed but not yet terminated, i.e., T'CALLABLE can be FALSE when T'TERMINATED is not yet TRUE.

AI-00444/05: Conditional entry calls can be queued momentarily [9.7.2(01)]

A conditional entry call may (momentarily) increase the COUNT attribute of an entry, even if the conditional call is not accepted.

AI-00446/05: Raising an exception in an abnormally completed task [9.10(06), 11.4(01)]

An exception can be propagated to an abnormally completed task that is engaged in a rendezvous or that is waiting for a task to be activated. If this occurs, the exception has no effect.

AI-00449/04: Evaluating default discriminant expressions [3.3.2(06)]

Default discriminant expressions are not evaluated when a subtype indication is elaborated.

AI-00455/05: Raising an exception before the sequence of statements [11.4.1(03)]

If an exception is raised due to the attempt to activate a task and the exception is raised after elaboration of a declarative part and just before execution of a sequence of statements, the sequence of statements is not executed and control is transferred in the same manner as for an exception raised in the sequence of statements.

AI-00464/05: Delay statements executed by the environment task [9.6(01)]

Delay statements can be executed by the environment task when a library package is elaborated. Such statements delay the environment task.

AI-00466/04: I/O performed by library tasks [14.1(07)]

The language does not define what happens to external files after the completion of the main program and before completion of all the library tasks.

AI-00467/04: Correction to AI-00179/06 [3.5.10(08)]

In the discussion section of AI-00179/06, the upper bound of SG's range constraint and the model interval in the subsequent discussion are incorrect because the model numbers for subtype SG are the same as the model numbers for type G.

AI-00468/04: Correction to AI-00187/04 discussion [4.1.3(15)]

The discussion section of AI-00187/04 should be corrected to point out that a name declared by a renaming declaration is not allowed as the prefix of an expanded name if the selector is not declared in the visible part of the package denoted by the prefix.

AI-00471/04: Correction to AI-00144/08 examples [3.5.9(09)]

In the examples, the value of **delta** is given incorrectly as an integer expression, $2**(-15)$; the expression should be $2.0**(-15)$.

AI-00475/05: Multiplication of fixed point values by negative integers [4.5.5(08)]

If the integer in an integer multiplication of a fixed point value is negative, the multiplication is equivalent to changing the sign of the fixed point value followed by repeated addition.

AI-00483/04: Correction to question in AI-00409/03 [12.3(05)]

The instantiation of SET_OF in the question is illegal because the formal generic type is an integer type. The formal type should be declared as a discrete type.

AI-00486/04: Correction to AI-00047/06 example [14.3.1(04)]

The call to SET_PAGE_LENGTH in the question's example should refer to file FT, not the default output file.

AI-00493/05: Operator symbols that represent the same operator [6.3.1(04)]

Two string literals serving as operator symbols represent the same operator if the string literals are identical or if the only difference is that some letters appear in upper case rather than lower case.

AI-00502/05: Error in AI-00170/06 [8.5(05)]

The declaration of type SINT in the question's example should be replaced with a subtype declaration so SINT has the type INTEGER.

AI-00503/04: Error in AI-00258/05 [13.7.2(07)]

The renaming declaration for Ren_C2 in AI-00258/05 is illegal since a type mark is required instead of a subtype indication. An appropriate subtype declaration should be added to the example.

AI-00508/03: The safe numbers of a fixed point subtype [3.5.9(11)]

The safe numbers of a fixed point subtype are the safe numbers of its base type.

AI-00511/05: Error in AI-00388/04 [2.8(04)]

The example given in the response is syntactically incorrect unless "**package** P" is replaced with either "**procedure** P" or "**package** P **is end**".

AI-00516/05: The safe interval for a fixed/integer result [4.5.7(04)]

When a fixed point value is divided by an integer value, the result model interval is determined by considering the integer value to be a model interval consisting of a single integer value.

AI-00866/03: The Latin-1 character set is used in source code and literals [2.1(01), 2.5(01), 2.7(01), C(13)]

The Ada character set is based on *ISO* 8859/1, Latin-1. This commentary replaces and extends AI-00339, which applied only to character codes used in comments.

An implementation is allowed to provide pragmas or operating modes in which the graphic characters used in identifiers and string literals are defined in a locally appropriate manner (e.g., Latin-2).

Index

An entry exists in this index for each technical term or phrase that is defined in the reference manual. The term or phrase is in boldface and is followed by the section number where it is defined, also in boldface, for example:

Record aggregate 4.3.1

References to other sections that provide additional information are shown after a semicolon, for example:

Record aggregate 4.3.1; 4.3

References to other related entries in the index follow in brackets, and a line that is indented below a boldface entry gives the section numbers where particular uses of the term or phrase can be found; for example:

Record aggregate 4.3.1; 4.3
    [see also: aggregate]
        as a basic operation 3.3.3; 3.7.4
        in a code statement 13.8

The index also contains entries for different parts of a phrase, entries that correct alternative terminology, and entries directing the reader to information otherwise hard to find, for example:

Check
    [see: suppress pragma]

The entries for the DEC Ada technical terms, phrases, references, and so on have been incorporated into the index following the established conventions. Additions to an existing entry have been added to the end of the entry; additional main entries have been inserted directly into the existing index. All DEC Ada additions are distinguished by colored print.

Abandon elaboration or evaluation (of declarations or statements)
   [see: exception, raise statement]

Abnormal task 9.10; 9.9
   [see also: abort statement]
      as recipient of an entry call 9.7.2, 9.7.3, 11.5; 9.5
      raising tasking_error in a calling task 11.5; 9.5

Abort statement 9.10
   [see also: abnormal task, statement, task]
      as a simple statement 5.1

Abs unary operator 4.5.6; 4.5
   [see also: highest precedence operator]
      as an operation of a fixed point type 3.5.10
      as an operation of a floating point type 3.5.8
      as an operation of an integer type 3.5.5
      in a factor 4.4

Absolute global symbol D; 13.9a.2.1, 13.9a.2.2, 13.9a.2.3

Absolute value operation 4.5.6

Accept alternative (of a selective wait) **9.7.1**
      for an interrupt entry 13.5.1

Accept statement 9.5; 9, D
   [see also: entry call statement, simple name in . . . , statement, task]
      accepting a conditional entry call 9.7.2
      accepting a timed entry call 9.7.3
      and optimization with exceptions 11.6
      as a compound statement 5.1
      as part of a declarative region 8.1
      entity denoted by an expanded name 4.1.3
      in an abnormal task 9.10

      in a select alternative 9.7.1
      including an exit statement 5.7
      including a goto statement 5.9
      including a return statement 5.8
      raising an exception 11.5
      to communicate values 9.11

Access to external files **14.2**

Access type 3.8; 3.3, D
   [see also: allocator, appropriate for a type, class of type, collection, derived type of an access type, null access value, object designated by . . .]
      as a derived type 3.4
      as a generic formal type 12.1.2, 12.3.5
      deallocation [see: unchecked_deallocation]
      designating a limited type 7.4.4
      designating a task type determining task dependence 9.4
      formal parameter 6.2
      name in a controlled pragma 4.8
      object initialization 3.2.1
      operation 3.8.2
      prefix 4.1
      value designating an object 3.2, 4.8
      value designating an object with discriminants 5.2
      with a discriminant constraint 3.7.2
      with an index constraint 3.6.1
      input-output of 14.1

Access type definition 3.8; 3.3.1, 12.1.2
      as a generic type definition 12.1

Access_check
   [see: constraint_error, suppress]
   [see also: address (predefined attribute); bit (DEC Ada predefined attribute)]

Accuracy
> of a numeric operation 4.5.7
> of a numeric operation of a
> universal type 4.10

Activation
> [see: task activation]

Actual object
> [see: generic actual object]

Actual parameter 6.4.1; D; (of an
operator) 6.7; (of a subprogram) 6.4; 6.2,
6.3
> [see also: entry call, formal
> parameter, function call, procedure
> call statement, subprogram call]
>> characteristics and overload
>> resolution 6.6
>> in a generic instantiation
> [see: generic actual parameter]
>> of an array type 3.6.1
>> of a record type 3.7.2
>> of a task type 9.2
>> that is an array aggregate 4.3.2
>> that is a loop parameter 5.5

Actual parameter part 6.4
> in a conditional entry call 9.7.2
> in an entry call statement 9.5
> in a function call 6.4
> in a procedure call statement 6.4
> in a timed entry call 9.7.3

Actual part
> [see: actual parameter part, generic
> actual part]

Actual subprogram
> [see: generic actual subprogram]

Actual type
> [see: generic actual type]

ADD_ATOMIC (DEC Ada predefined
procedure)
> [see:system.add_atomic]

Adding operator
> [see: binary adding operator, unary
> adding operator]

ADD_INTERLOCKED (DEC Ada
predefined procedure)
> [see:system.add_interlocked]

Addition operation 4.5.3
> accuracy for a real type 4.5.7

ADDRESS (predefined attribute) **13.7.2**;
3.5.5, 3.5.8, 3.5.10, 3.6.2, 3.7.4, 3.8.2, 7.4.2,
9.9, 13.7, A
> [see also: address clause,
> system.address]

ADDRESS (predefined type)
> [see: system.address]

Address clause 13.5; 13.1, 13.7
> [see also: storage address,
> system.address]
>> as a representation clause 13.1
>> for an entry 13.5.1
>> DEC Ada interpretation of
>> expressions F.7
>> DEC Ada restrictions on 13.5

ADDRESS_OPERATIONS (DEC Ada
predefined package) **13.7a.1**

ADDRESS_SIZE (DEC Ada predefined
constant)
> [see:system.address_size]

ADDRESS_ZERO (DEC Ada predefined
constant)
> [see: system.address_zero]

AFT (predefined attribute) for a fixed
point type **3.5.10**; A

Aft field of text_io output **14.3.8**, **14.3.10**

Aggregate 4.3, **D**
> [see also: array aggregate,
> overloading of . . . , record aggregate]
>> as a basic operation 3.3.3; 3.6.2,
>> 3.7.4
>> as a primary 4.4

in an allocator 4.8
in a code statement 13.8
in an enumeration representation
clause 13.3
in a qualified expression 4.7
must not be the argument of a
conversion 4.6
of a derived type 3.4

ALIGNED_WORD (DEC Ada predefined
type)
[see: system.aligned_word]

Alignment clause (in a record
representation clause) **13.4**

All in a selected component **4.1.3**

Allocation
of array and record components
13.1, 13.4
of variables forced to memory
with representation attribute
address 13.7.2

Allocation of processing
resources **9.8**

Allocator 4.8; 3.8, D
[see also: access type, collection,
exception raised during . . . , initial
value, object, overloading of . . .]
as a basic operation 3.3.3; 3.8.2
as a primary 4.4
creating an object with a
discriminant 4.8; 5.2
for an array type 3.6.1
for a generic formal access type
12.1.2
for a private type 7.4.1
for a record type 3.7.2
for a task type 9.2; 9.3
must not be the argument of a
conversion 4.6
raising storage_error due to
the size of the collection being
exceeded 11.1
setting a task value 9.2
without storage check 11.7

Allowed 1.6

Alternate key in an indexed access file
14.2a

Alternative
[see: accept alternative, case
statement alternative, closed
alternative, delay alternative,
open alternative, select alternative,
selective wait, terminate alternative]

Ambiguity
[see: overloading]

Ampersand
[see: catenation]
character 2.1
delimiter 2.2

Ancestor library unit 10.2

And operator
[see: logical operator]

And then control form
[see: short circuit control form]

AND_ATOMIC (DEC Ada predefined
procedure)
[see:system.and_atomic]

Anonymous type 3.3.1; 3.5.4, 3.5.7,
3.5.9, 3.6, 9.1

Anonymous base type
[see: first named subtype]

ANSI (american national standards
institute) **2.1**

Apostrophe character 2.1
in a character literal 2.5

Apostrophe delimiter 2.2
in an attribute 4.1.4
of a qualified expression 4.7

Apply 10.1.1

Appropriate for a type **4.1**
> for an array type 4.1.1, 4.1.2
> for a record type 4.1.3
> for a task type 4.1.3

Arbitrary selection of select alternatives **9.7.1**

Argument association in a pragma **2.8**

Argument identifier in a pragma **2.8**

Arithmetic operator 4.5
> [see also: binary adding operator, exponentiating operator, multiplying operator, predefined operator, unary adding operator]
>> as an operation of a fixed point type 3.5.10
>> as an operation of a floating point type 3.5.8
>> as an operation of an integer type 3.5.5
>> rounding for real types 13.7.3

Array aggregate 4.3.2; 4.3
> [see also: aggregate]
>> as a basic operation 3.3.3; 3.6.2
>> in an enumeration representation clause 13.3

Array assignment 5.2.1

Array bounds
> [see: bound of an array]

Array component
> [see: array type, component, indexed component]
> [see also: allocation]

Array type 3.6; 3.3, D
> [see also: component, composite type, constrained array, constrained . . . , index, matching components, null slice, slice, unconstrained . . .]
>> as a full type 7.4.1
>> as a generic formal type 12.1.2
>> as a generic parameter 12.3.4

> as the type of a formal parameter 6.2
> conversion 4.6
> for a prefix of an indexed component 4.1.1
> for a prefix of a slice 4.1.2
> operation 3.6.2; 4.5.2, 4.5.3
> operation on an array of boolean components 4.5.1, 4.5.6
> with a component type with discriminants 3.7.2
> with a limited component type 7.4.4
> maximum number of dimensions in DEC Ada F

Array type definition 3.6; 3.3.1, 12.1.2, 12.3.4
> [see also: constrained array definition, elaboration of . . . , unconstrained array definition]
>> as a generic type definition 12.1

Arrow compound delimiter 2.2

ASCII (american standard code for information interchange) **2.1**

ASCII (predefined library package) **3.5.2**; 2.6, C
> [see also: graphical symbol]

Assignment compound delimiter 2.2; 5.2
> in an object declaration 3.2.1

Assignment operation 5.2; D
> [see also: initial value, limited type]
>> as a basic operation 3.3, 3.3.3; 3.5.5, 3.5.8, 3.5.10, 3.6.2, 3.7.4, 3.8.2, 7.4.2, 12.1.2
>> for a generic formal type 12.1.2
>> not available for a limited type 7.4.4
>> of an array aggregate 4.3.2
>> of an initial value to an object 3.2.1
>> to an array variable 5.2.1; 5.2
>> to a loop parameter 5.5

to an object designated by an
access value 3.8
to a shared variable 9.11

Assignment statement 5.2; D
[see also: statement]
as a simple statement 5.1

ASSIGN_TO_ADDRESS (DEC Ada generic procedure)
[see: system.assign_to_address]

Associated declarative region of a declaration or statement **8.1**

Association
[see: component association, discriminant association, generic association, parameter association]

AST_ENTRY (DEC Ada predefined attribute) **9.12a**; A

AST_ENTRY (DEC Ada predefined pragma) **9.12a**; B

AST_HANDLER (DEC Ada predefined type)
[see: system.ast_handler]

Asynchronous system trap (AST) (OpenVMS) D
[see also: ast_entry, system.ast_handler]

Attribute 4.1.4; D
[see also: predefined attribute, representation attribute]
as a basic operation 3.3.3
as a name 4.1
as a primary 4.4
in a length clause 13.2
in a static expression in a generic unit 12.1
of an access type 3.5.8
of an array type 3.6.2
of a derived type 3.4
of a discrete type or subtype 3.5.5
of an entry 9.9

of a fixed point type 3.5.10
of a floating point type 3.5.8
of an object of a task type 9.9
of a private type 7.4.2; 3.7.4
of a record type 3.7.4
of a static subtype in a static expression 4.9
of a task type 9.9
of a type 3.3
of a type as a generic actual function 12.3.6
of a type with discriminants 3.7.4
renamed as a function 8.5
that is a function 3.5.5

Attribute designator 4.1.4

AUX_IO_EXCEPTIONS (DEC Ada predefined input-output package) **14.4**; 14.2a.3, 14.2a.5, 14.2b.8, 14.2b.10
specification 14.5a

Bar
[see: vertical bar]

BASE (predefined attribute) **3.3.3**; A
for an access type 3.8.2
for an array type 3.6.2
for a discrete type 3.5.5
for a fixed point type 3.5.10
for a floating point type 3.5.8
for a private type 7.4.2
for a record type 3.7.4

Base type (of a subtype) **3.3**
as a static subtype 4.9
as target type of a conversion 4.6
due to elaboration of a type definition 3.3.1
name [see: name of a base type]
of an array type 3.6; 4.1.2
of a derived subtype 3.4

Blank skipped by a text_io procedure **14.3.5**

Block name 5.6
> declaration 5.1
> implicitly declared 3.1

Block statement 5.6; D
> [see also: completed block statement, statement]
>> as a compound statement 5.1
>> as a declarative region 8.1
>> entity denoted by an expanded name 4.1.3
>> having dependent tasks 9.4
>> including an exception handler 11.2; 11
>> including an implicit declaration 5.1
>> including a suppress pragma 11.7
>> raising an exception 11.4.1, 11.4.2

Body 3.9; D
> [see also: declaration, generic body, generic package body, generic subprogram body, library unit, package body, proper body, subprogram body, task body]
>> as a later declarative item 3.9

Body stub 10.2; D
>> acting as a subprogram declaration 6.3
>> as a body 3.9
>> as a portion of a declarative region 8.1
>> must be in the same declarative region as the declaration 3.9, 7.1

BOOLEAN (predefined type) **3.5.3**; C
>> derived 3.4; 3.5.3
>> result of a condition 5.3
>> result of an explicitly declared equality operator 6.7

Boolean expression
> [see: condition, expression]

Boolean operator
> [see: logical operator]

Boolean type 3.5.3
> [see also: derived type of a boolean type, predefined type]
>> operation 3.5.5; 4.5.1, 4.5.2, 4.5.6
>> operation comparing real operands 4.5.7

Bound
> [see: error bound, first attribute, last attribute]

Bound of an array **3.6**, **3.6.1**
> [see also: index range, slice]
>> aggregate 4.3.2
>> ignored due to index_check suppression 11.7
>> initialization in an allocator constrains the allocated object 4.8
>> that is a formal parameter 6.2
>> that is the result of an operation 4.5.1, 4.5.3, 4.5.6

Bound of a range **3.5**; 3.5.4
>> of a discrete range in a slice 4.1.2
>> of a discrete range is of universal_integer type 3.6.1
>> of a static discrete range 4.9

Bound of a scalar type **3.5**

Bound of a slice **4.1.2**

Box D

Box compound delimiter 2.2
>> in a generic parameter declaration 12.1, 12.1.2, 12.1.3; 12.3.3
>> in an index subtype definition 3.6

Bracket
> [see: label bracket, left parenthesis, parenthesized expression, right parenthesis, string bracket]

CALENDAR (predefined library package) **9.6**; C

Call
    [see: conditional entry call, entry call statement, function call, procedure call statement, subprogram call, timed entry call]

CALLABLE (predefined attribute)
      for an abnormal task 9.10
      for a task object 9.9; A

Calling conventions
    [see: subprogram declaration]
      of a subprogram written in another language 13.9

Cancelation of an entry call statement **9.7.2**, **9.7.3**

Carriage return format effector **2.1**

Case of a letter
    [see: letter, lower case letter, upper case letter]

Case statement 5.4
    [see also: statement]
      as a compound statement 5.1

Case statement alternative 5.4

Catenation operation 4.5.3
      for an array type 3.6.2
      in a replacement of a string literal 2.10

Catenation operator 4.5; 2.6, 3.6.3, 4.5.3, C
    [see also: predefined operator]

Cell in a relative access file **14.2a**

Character 2.1
    [see also: ampersand, apostrophe, basic character, colon, divide, dot, equal, exclamation mark character, graphic character, greater than, hyphen, less than, minus, other special character, parenthesis, percent, period, plus, point character, pound sterling, quotation, semicolon, sharp, space, special character, star, underline, vertical bar]
      in a lexical element 2, 2.2
      names of characters 2.1
      replacement in program text 2.10
      image of a nongraphic 3.5.5

CHARACTER (predefined type) **3.5.2**; C
      as the component type of the type string 3.6.3

Character literal 2.5; 3.5.2, 4.2
    [see also: scope of . . . , space character literal, visibility of . . .]
      as a basic operation 3.3.3
      as an enumeration literal 3.5.1
      as a name 4.1
      as a selector 4.1.3
      declared by an enumeration literal specification 3.1
      in a static expression 4.9
      in homograph declarations 8.3
      must be visible at the place of a string literal 4.2

Character type 3.5.2; 2.5
      operation 3.5.5

Check
    [see: suppress pragma]

Choice 3.7.3
    [see also: exception choice]
      in an aggregate 4.3
      in an array aggregate 4.3.2
      in a case statement alternative 5.4
      in a component association 4.3, 4.3.1, 4.3.2
      in a record aggregate 4.3.1
      in a variant of a record type definition 3.7.3

Circularity in dependences
      between compilation units **10.5**

Class of type 3.3; 12.1.2
[see also: access type, composite type, private type, scalar type, task type]
of a derived type 3.4

Clause
[see: address clause, alignment clause, component clause, context clause, enumeration representation clause, length clause, record representation clause, representation clause, use clause, with clause]

CLEAR_INTERLOCKED (DEC Ada predefined procedure)
[see: system.clear_interlocked]

CLOCK (predefined function) **9.6**
[see also: system.tick]

CLOSE (input-output procedure)
in an instance of direct_io 14.2.1; 14.2.5
in an instance of sequential_io 14.2.1; 14.2.3
in text_io 14.2.1; 14.3.10
in an instance of indexed_io 14.2.1; 14.2a.5
in an instance of relative_io 14.2a.1; 14.2a.3
in direct_mixed_io 14.2.1; 14.2b.6
in indexed_mixed_io 14.2.1; 14.2b.10
in relative_mixed_io 14.2.1; 14.2b.8
in sequential_mixed_io 14.2.1; 14.2b.4

Closed alternative (of a selective wait) **9.7.1**; 11.1
[see also: alternative]

Closed file 14.1

Code statement 13.8
[see also: statement]
as a simple statement 5.1

COL (text_io function) **14.3.4**; 14.3.10
raising an exception 14.4

Collection (of an access type) **3.8**; 4.8, D
[see also: access type, allocator, length clause, object, storage units allocated, storage_size attribute]
of a derived access type 13.2; 3.4

Colon character 2.1
[see also: based literal]
replacing sharp character 2.10

Colon delimiter 2.2

Column 14.3.4

Comma
character 2.1
delimiter 2.2

Comment 2.7; 2.2
in a conforming construct 6.3.1

COMMON_OBJECT (DEC Ada predefined pragma) **13.9a.2.3**; 13.9a, 13.9a.2, B
maximum number of objects declared with F

Communication
between tasks [see: accept statement, entry, rendezvous]
of values between tasks 9.5, 9.11

Comparison
[see: relational operator]

Compatibility (of constraints) **3.3.2**
[see also: constraint]
failure not causing constraint_ error 11.7
of a discrete range with an index subtype 3.6.1
of discriminant constraints 3.7.2
of fixed point constraints 3.5.9
of floating point constraints 3.5.7
of index constraints 3.6.1
of range constraints 3.5

Compilation 10.1; 10, 10.4

    as a sequence of lexical elements 2

    including an inline pragma 6.3.2

Compilation order

    [see: order of compilation]

Compilation unit 10.1; 10, 10.4, D

    [see also: library unit, secondary unit]

        compiled after library units named in its context clause 10.3

        followed by an inline pragma 6.3.2

        with a context clause 10.1.1

        with a use clause 8.4

Compile time evaluation of expressions **10.6**; 4.9

Compiler 10.4

Compiler listing

    [see: list pragma, page pragma]

Compiler optimization

    [see: optimization, optimize pragma]

Completed block statement 9.4

Completed subprogram 9.4

Completed task 9.4; 9.9

    [see also: tasking_error, terminated task]

        as recipient of an entry call 9.5, 9.7.2, 9.7.3

        becoming abnormal 9.10

        completion during activation 9.3

        due to an exception in the task body 11.4.1, 11.4.2

Component (of a composite type) **3.3**; 3.6, 3.7, D

    [see also: component association, component clause, component list, composite type, default expression, dependence on a discriminant, discriminant, indexed component, object, record type, selected component, subcomponent]

        combined by aggregate 4.3

        depending on a discriminant 3.7.1; 11.1

        name starting with a prefix 4.1

        of an array 3.6 [see also: array type]

        of a constant 3.2.1

        of a derived type 3.4

        of an object 3.2

        of a private type 7.4.2

        of a record 3.7 [see also: record type]

        of a variable 3.2.1

        simple name as a choice 3.7.3

        subtype 3.7

        subtype itself a composite type 3.6.1, 3.7.2

        that is a task object 9.3

        whose type is a limited type 7.4.4

        biasing of 13.4

Component association 4.3

        in an aggregate 4.3

        including an expression that is an array aggregate 4.3.2

        named component association 4.3

        named component association for selective visibility 8.3

        positional component association 4.3

Component clause (in a record representation clause) **13.4**

Component declaration 3.7

    [see also: declaration, record type definition]

        as part of a basic declaration 3.1

        having an extended scope 8.2

        in a component list 3.7

        of an array object 3.6.1

        of a record object 3.7.2

        visibility 8.3

Constant declaration 3.2.1
[see also: deferred constant
declaration]
as a full declaration 7.4.3
with an array type 3.6.1
with a record type 3.7.2

CONSTRAINED (predefined attribute)
for an object of a type with
discriminants 3.7.4; A
for a private type 7.4.2, A

Constrained array definition 3.6
in an object declaration 3.2, 3.2.1

Constrained array type 3.6
[see also: array type, constraint]

Constrained subtype 3.3; 3.2.1, 3.6,
3.6.1, 3.7, 3.7.2, 6.4.1, 12.3.4
[see also: constraint, subtype, type,
unconstrained subtype]
due to elaboration of a type
definition 3.3.1
due to the elaboration of a derived
type definition 3.4
object declarations 3.2.1
of a subtype indication in an
allocator 4.8

Constraint (on an object of a type) **3.3**,
3.3.2; D
[see also: accuracy constraint,
compatibility, constrained subtype,
dependence on a discriminant,
discriminant constraint, elaboration
of . . . , fixed point constraint, floating
point constraint, index constraint,
range constraint, satisfy, subtype,
unconstrained subtype]
explicitly specified by use of a
qualification 4.7
in a subtype indication in an
allocator 4.8
not considered in overload
resolution 8.7
on a derived subtype 3.4
on a formal parameter 6.2

on a formal parameter of a
generic formal subprogram 12.1.3
on a generic actual parameter
12.3.1
on a generic formal object 12.1.1
on a generic formal parameter
12.1; 12.3.1
on an object designated by an
access value 3.8
on a renamed object 8.5
on a subcomponent subject to a
component clause must be static
13.4
on a subtype of a generic formal
type 12.1.2
on a type mark in a generic
parameter declaration 12.3.1
on a variable 3.2.1, 3.3, 3.6
on the result of a generic formal
function 12.1.3

CONSTRAINT_ERROR (predefined
exception) **11.1**
[see also: suppress pragma]
raised by an accept statement 9.5
raised by an actual parameter
not in the subtype of the formal
parameter 6.4.1
raised by an allocator 4.8
raised by an assignment 5.2; 3.5.4
raised by an attribute 3.5.5
raised by a component of an array
aggregate 4.3.2
raised by a component of a record
aggregate 4.3.1
raised by an entry call statement
9.5
raised by a formal parameter
not in the subtype of the actual
parameter 6.4.1
raised by an index value out of
bounds 4.1.1, 4.1.2
raised by a logical operation on
arrays of different lengths 4.5.1
raised by a name with a prefix
evaluated to a null access value
4.1

Default initialization (for an object) **3.2.1**, **3.3**

    [see also: default expression, default initial value, initial value]

Default mode (of a file) **14.2.1**; 14.2.3, 14.2.5, 14.3.10

Default_aft (field length) of fixed_io or float_io 14.3.8; 14.3.10

Default_base of integer_io 14.3.7; 14.3.10

Default_exp (field length) of fixed_io or float_io 14.3.8; 14.3.10

Default_fore (field length) of fixed_io or float_io 14.3.8; 14.3.10

Default_setting (letter case) of enumeration_io 14.3.9; 14.3.10

Default_width (field length)
    of enumeration_io 14.3.9; 14.3.10
    of integer_io 14.3.7; 14.3.10

Deferred constant 7.4.3
    of a limited type 7.4.4

Deferred constant declaration 7.4; 7.4.3

    [see also: private part (of a package), visible part (of a package)]
        as a basic declaration 3.1
        is not a forcing occurrence 13.1

Definition

    [see: access type definition, array type definition, component subtype definition, constrained array definition, derived type definition, enumeration type definition, generic type definition, index subtype definition, integer type definition, real type definition, record type definition, type definition, unconstrained array definition]

Delay alternative (of a selective wait) **9.7.1**

Delay expression 9.6; 9.7.1

    [see also: duration]
        in a timed entry call 9.7.3

Delay statement 9.6

    [see also: statement, task]
        as a simple statement 5.1
        in an abnormal task 9.10
        in a select alternative 9.7.1
        in a timed entry call 9.7.3

DELETE (input-output procedure)
        in an instance of direct_io 14.2.1; 14.2.5
        in an instance of sequential_io 14.2.1; 14.2.3
        in text_io 14.2.1; 14.3.10
        in an instance of indexed_io 14.2.1; 14.2a.5
        in an instance of relative_io 14.2.1; 14.2a.3
        in direct_mixed_io 14.2.1; 14.2b.6
        in indexed_mixed_io 14.2.1; 14.2b.10
        in relative_mixed_io 14.2.1; 14.2b.8
        in sequential_mixed_io 14.2.1; 14.2b.4

DELETE_ELEMENT (DEC Ada input-output procedure)
        in an instance of indexed_io 14.2a.4, 14.2a.5
        in an instance of relative_io 14.2a.2, 14.2a.3
        in indexed_mixed_io 14.2b.9, 14.2b.10
        in relative_mixed_io 14.2b.7, 14.2b.8

Delimiter 2.2

    [see also: ampersand, apostrophe, arrow, assignment, colon, compound delimiter, divide, dot, double dot, equal, exclamation mark, exponentiation, greater than or equal, greater than, inequality, label bracket, less than or equal, less than,

minus, parenthesis, period, plus, point, semicolon, star, vertical bar]

Delta (of a fixed point type) **3.5.9**
[see also: fixed point type]
of universal_fixed 4.5.5

DELTA (predefined attribute) **3.5.10**; 4.1.4, A

Denote an entity 3.1, **4.1**; D
[see also: declaration, entity, name]

Dependence between compilation units **10.3**; 10.5
[see also: with clause]
circularity implying illegality 10.5

Dependence on a discriminant **3.7.1**; 3.7
[see also: component subtype definition, component, constraint, discriminant constraint, discriminant, index constraint, subcomponent, subtype definition, variant part]
affecting renaming 8.5
by a subcomponent that is an actual parameter 6.2
effect on compatibility 3.7.2
effect on matching of components 4.5.2
for an assignment 5.2

Dependent task 9.4
delaying exception propagation 11.4.1
of an abnormal task 9.10

Derivable subprogram 3.4
prohibiting representation clauses 13.1

Derived subprogram 3.4
as an operation 3.3.3
implicitly declared 3.3.3

Derived type 3.4; D
[see also: parent type]
conversion to or from a parent type or related type 4.6

of an access type [see: access type, collection]
of an access type designating a task type determining task dependence 9.4
of a boolean type 3.4, 3.5.3
of a limited type 7.4.4
of a private type 7.4.1
subject to a representation clause 13.1, 13.6

Derived type definition 3.4; 3.3.1
[see also: elaboration of . . .]

Descriptor D; 13.9a.1.4

Descriptor parameter passing mechanism **13.9a.1.4**

Designate 3.8, **9.1**; D
[see also: access type, allocator, object designated by . . . , task designated by . . . , task object designated by . . .]

Designated subtype (of an access type) **3.8**

Designated type (of an access type) **3.8**

Designator (of a function) **6.1**
[see also: attribute designator, operator, overloading of . . .]
in a function declaration 4.5
in a subprogram body 6.3
in a subprogram specification 6.1; 6.3
of a generic formal subprogram 12.3.6; 12.1, 12.1.3
of a library unit 10.1
overloaded 6.6

DEVICE_ERROR (input-output exception) **14.4**; 14.2.3, 14.2.5, 14.3.10, 14.5, 14.2a.3, 14.2a.5, 14.2b.4, 14.2b.6, 14.2b.8, 14.2b.10

D_FLOAT (DEC Ada predefined type)
[see system.d_float]

D_floating (VAX floating point type representation) **3.5.7**; 3.5.7a, 3.5.7b
> values of for machine-dependent attributes F

Digit 2.1
> [see also: basic graphic character, extended digit, letter or digit]
>> in a based literal 2.4.2
>> in a decimal literal 2.4.1
>> in an identifier 2.3

Digits (of a floating point type) **3.5.7**
> [see also: floating point type]

DIGITS (predefined attribute) **3.5.8**; 4.1.4, A

Dimensionality of an array **3.6**

Direct access file 14.2; 14.1, 14.2.1

Direct input-output 14.2.4; 14.2.1

Direct visibility 8.3; D
> [see also: basic operation, character literal, operation, operator symbol, selected component, visibility]
>> due to a use clause 8.4
>> of a library unit due to a with clause 10.1.1
>> within a subunit 10.2

DIRECT_IO (predefined input-output generic package) **14.2**, **14.2.4**; 14, 14.1, 14.2.5, C
>> exceptions 14.4; 14.5
>> specification 14.2.5
>> requisite specification of FORM parameter with 14.1b

DIRECT_MIXED_IO (DEC Ada predefined input-output package) **14.2b.5**; 14.2b.6
>> exceptions 14.4; 14.5
>> requisite specification of FORM parameter with 14.1b
>> specification 14.2b.6

Discrete range 3.6, **3.6.1**
> [see also: range, static discrete range]
>> as a choice 3.7.3
>> as a choice in an aggregate 4.3
>> for a loop parameter 5.5
>> in a choice in a case statement 5.4
>> in a generic formal array type declaration 12.1.2; 12.3.4
>> in an index constraint 3.6
>> in a loop parameter specification 5.5
>> in a slice 4.1.2
>> of entry indices in an entry declaration 9.5

Discrete type 3.5; D
> [see also: basic operation of . . . , enumeration type, index, integer type, iteration scheme, operation of . . . , scalar type]
>> as a generic actual parameter 12.3.3
>> as a generic formal type 12.1.2
>> expression in a case statement 5.4
>> of a discriminant 3.7.1
>> of a loop parameter 5.5
>> of index values of an array 3.6
>> operation 3.5.5; 4.5.2

Discriminant 3.3, **3.7.1**; 3.7, D
> [see also: component clause, component, composite type, default expression, dependence on . . . , record type, selected component, subcomponent]
>> in a record aggregate 4.3.1
>> initialization in an allocator constrains the allocated object 4.8
>> of a derived type 3.4
>> of a formal parameter 6.2
>> of a generic actual type 12.3.2
>> of a generic formal type 12.3, 12.3.2
>> of an implicitly initialized object 3.2.1

of an object designated by an access value 3.7.2; 5.2
of a private type 7.4.2; 3.3
of a variant part must not be of a generic formal type 3.7.3
simple name in a variant part 3.7.3
subcomponent of an object 3.2.1
with a default expression 3.7.1; 3.2.1
maximum number in record type F

Discriminant association 3.7.2
in a discriminant constraint 3.7.2
named discriminant association 3.7.2
named discriminant association for selective visibility 8.3
positional discriminant association 3.7.2

Discriminant constraint 3.7.2; 3.3.2, D
[see also: dependence on a discriminant]
ignored due to access_check suppression 11.7
in an allocator 4.8
on an access type 3.8
violated 11.1

Discriminant part 3.7.1; 3.7
[see also: elaboration of . . .]
absent from a record type declaration 3.7
as a portion of a declarative region 8.1
conforming to another 3.8.1, 6.3.1, 7.4.1
in a generic formal type declaration 3.7.1; 12.1
in an incomplete type declaration 3.8.1
in a private type declaration 7.4, 7.4.1
in a type declaration 3.3, 3.3.1
must not include a pragma 2.8

of a full type declaration is not elaborated 3.3.1

Discriminant specification 3.7.1
[see also: default expression]
as part of a basic declaration 3.1
declaring a component 3.7
having an extended scope 8.2
in a discriminant part 3.7.1
visibility 8.3

Discriminant_check
[see: constraint_error, suppress]
[see also: address (predefined attribute), bit (DEC Ada predefined attribute), size (predefined attribute)]

Disjunction
[see: logical operator]

Divide
character 2.1
delimiter 2.2

Division operation 4.5.5
accuracy for a real type 4.5.7

Division operator
[see: multiplying operator]

Division_check
[see: numeric_error, suppress]

Dot
[see: double dot]
character 2.1 [see also: double dot, point character]
delimiter 2.2
delimiter of a selected component 8.3; 4.1.3

Double dot compound delimiter **2.2**

Double hyphen starting a comment **2.7**

Double star compound delimiter **2.2**
[see also: exponentiation compound delimiter]

DURATION (predefined type) **9.6**; C
[see also: delay expression, fixed point type]
of alternative delay statements 9.7.1
DEC Ada values for F

Effect
[see: elaboration has no other effect]

ELABORATE (predefined pragma) **10.5**; B

Elaborated 3.9

Elaboration 3.9; 3.1, 3.3, 10.1, D
[see also: exception raised during . . . , order of elaboration]
optimized 10.6

Elaboration has no other effect 3.1

Elaboration of
an access type definition 3.8
an array type definition 3.6
a body stub 10.2
a component declaration 3.7
a component subtype definition 3.7
a constrained array definition 3.6
a declaration 3.1
a declarative item 3.9
a declarative part 3.9
a deferred constant declaration 7.4.3
a derived type definition 3.4
a discriminant constraint 3.7.2
a discriminant part 3.7.1
a discriminant specification 3.7.1
an entry declaration 9.5
an enumeration literal specification 3.5.1
an enumeration type definition 3.5.1

a fixed point type declaration 3.5.9
a floating point type declaration 3.5.7
a formal part 6.1
a full type declaration 3.3.1
a generic body 12.2
a generic declaration 12.1
a generic instantiation 12.3
an incomplete type declaration 3.8.1
an index constraint 3.6.1
an integer type definition 3.5.4
a library unit 10.5
a loop parameter specification 5.5
an object declaration 3.2.1
a package body 7.3
a package declaration 7.2
a parameter specification 6.1
a private type declaration 7.4.1
a range constraint 3.5
a real type definition 3.5.6
a record type definition 3.7
a renaming declaration 8.5
a representation clause 13.1
a subprogram body 6.3
a subprogram declaration 6.1
a subtype declaration 3.3.2
a subtype indication 3.3.2
a task body 9.1
a task declaration 9.1
a task specification 9.1
a type declaration 3.3.1, 3.8.1, 7.4.1
a type definition 3.3.1
an unconstrained array definition 3.6
a use clause 8.4

Elaboration_check
[see: program_error exception, suppress]

Element in a file **14**, **14.1**; 14.2
in a direct access file 14.2.4
in a sequential access file 14.2.2
DEC Ada implementation of 14.1a

ELEMENT_SIZE (DEC Ada mixed-type input-output function)
> in direct_mixed_io 14.2b.2, 14.2b.6
> in indexed_mixed_io 14.2b.2, 14.2b.10
> in relative_mixed_io 14.2b.2, 14.2b.8
> in sequential_mixed_io 14.2b.2, 14.2b.4

ELEMENT_TYPE (generic formal type of direct_io) **14.2.5**; 14.1, 14.2.4

ELEMENT_TYPE (generic formal type of indexed_io) **14.2a.5**; 14.2a.4

ELEMENT_TYPE (generic formal type of relative_io) **14.2a.3**; 14.2a.2

ELEMENT_TYPE (generic formal type of sequential_io) **14.2.3**; 14.1, 14.2.2

Else part
> of a conditional entry call 9.7.2
> of an if statement 5.3
> of a selective wait 9.7.1; 11.1

EMAX (predefined attribute) **3.5.8**; A
> [see also: machine_emax]
> DEC Ada floating point values for F

Emin
> [see: machine_emin]

Empty string literal 2.6

End of line 2.2
> as a separator 2.2
> due to a format effector 2.2
> terminating a comment 2.7

END_ERROR (input-output exception) **14.4**; 14.2.2, 14.2.3, 14.2.4, 14.2.5, 14.3.4, 14.3.5, 14.3.6, 14.3.10, 14.5, 14.2a.3, 14.2a.4, 14.2a.5, 14.2b.3, 14.2b.4, 14.2b.5, 14.2b.6, 14.2b.8, 14.2b.9, 14.2b.10

END_OF_BUFFER (DEC Ada mixed-type input-output function)
> in direct_mixed_io 14.2b.2; 14.2b.6
> in indexed_mixed_io 14.2b.2; 14.2b.10
> in relative_mixed_io 14.2b.2; 14.2b.8
> in sequential_mixed_io 14.2b.2; 14.2b.4

END_OF_FILE (input-output function)
> in an instance of direct_io 14.2.4; 14.2.5
> in an instance of sequential_io 14.2.2; 14.2.3
> in text_io 14.3.1, 14.3.10
> in an instance of indexed_io 14.2a.4, 14.2a.5
> in an instance of relative_io 14.2a.2, 14.2a.3
> in direct_mixed_io 14.2b.5, 14.2b.6
> in indexed_mixed_io 14.2b.9, 14.2b.10
> in relative_mixed_io 14.2b.7, 14.2b.8
> in sequential_mixed_io 14.2b.3, 14.2b.4

END_OF_LINE (text_io function) **14.3.4**; 14.3.10
> raising an exception 14.4

END_OF_PAGE (text_io function) **14.3.4**; 14.3.10, 14.4

Entry (of a task) **9.5**; 9, 9.2, D
> [see also: actual parameter, address attribute, attribute of . . . , formal parameter, interrupt entry, overloading of . . . , parameter and result type profile, parameter, subprogram]
> > declared by instantiation of a generic formal parameter 12.3
> > denoted by an indexed component 4.1.1

Enumeration type 3.5.1; 3.3, 3.5, D
    [see also: discrete type, scalar type]
        as a character type 3.5.2
        as a generic formal type 12.1.2
        as a generic parameter 12.3.3
        boolean 3.5.3
        operation 3.5.5

Enumeration type definition 3.5.1;
3.3.1
    [see also: elaboration of . . .]

ENUMERATION_IO (text_io inner
generic package) **14.3.9**; 14.3.10

Environment of a program **10.4**
        environment task calling the
        main program 10.1

EPSILON (predefined attribute) **3.5.8**; A
        DEC Ada floating point values for
        F

Equal
        character 2.1
        delimiter 2.2

Equality operator 4.5; 4.5.2
    [see also: limited type, relational
operator]
        explicitly declared 4.5.2, 6.7; 7.4.4
        for an access type 3.8.2
        for an array type 3.6.2
        for a generic formal type 12.1.2
        for a limited type 4.5.2, 7.4.4
        for a real type 4.5.7
        for a record type 3.7.4

Erroneous execution 1.6
    [see also: program_error]
        due to an access to a deallocated
        object 13.10.1
        due to an unchecked conversion
        violating properties of objects of
        the result type 13.10.2
        due to assignment to a shared
        variable 9.11
        due to changing of a discriminant
        value 5.2, 6.2

        due to dependence on parameter
        passing mechanism 6.2
        due to multiple address clauses
        for overlaid entities 13.5
        due to suppression of an exception
        check 11.7
        due to use of an undefined value
        3.2.1
        due to an AST occurrence for an
        entry of a terminated task 9.12a

Error bounds of a predefined operation
of a real type **3.5.9**, **4.5.7**; 3.5.6, 3.5.7

Error detected at
        compilation time 1.6
        run time 1.6

Error situation 1.6, **11**, **11.1**; 11.6

Error that may not be detected **1.6**

Evaluation (of an expression) **4.5**; D
    [see also: compile time evaluation,
expression]
        at compile time 4.9, 10.6
        of an actual parameter 6.4.1
        of an aggregate 4.3; 3.3.3
        of an allocator 4.8
        of an array aggregate 4.3.2
        of a condition 5.3, 5.5, 5.7, 9.7.1
        of a default expression 3.7.2
        of a default expression for a
        formal parameter 6.4.2; 6.1
        of a discrete range 3.5; 9.5
        of a discrete range used in an
        index constraint 3.6.1
        of an entry index 9.5
        of an expression in an assignment
        statement 5.2
        of an expression in a constraint
        3.3.2
        of an expression in a generic
        actual parameter 12.3
        of an indexed component 4.1.1
        of an initial value [see: default
        expression]
        of a literal 4.2; 3.3.3
        of a logical operation 4.5.1

task activation 9.3

Exceptions and optimization 11.6

Exclamation character 2.1
  replacing vertical bar 2.10

Exclusive disjunction
  [see: logical operator]

Execution
  [see: sequence of statements, statement, task body, task]

EXISTENCE_ERROR (DEC Ada input-output exception) 14.4; 14.2a.2, 14.2a.3, 14.2a.4, 14.2a.5, 14.2b.7, 14.2b.8, 14.2b.9, 14.2b.10, 14.5a

Exit statement 5.7
  [see also: statement]
    as a simple statement 5.1
    causing a loop to be exited 5.5
    causing a transfer of control 5.1
    completing block statement execution 9.4

Expanded name 4.1.3; D
    denoting a loop 5.5
    in a static expression 4.9
    of a parent unit 10.2
    replacing a simple name 6.3.1

Explicit conversion 4.6
  [see also: conversion operation, implicit conversion, subtype conversion, type conversion]
    from universal_fixed type 4.5.5
    to a real type 4.5.7

Explicit declaration 3.1; 4.1
  [see also: declaration]

Explicit initialization
  [see: allocator, object declaration, qualified expression]

Exponent of a floating point number **3.5.7**; 13.7.3

Exponent part
    in output of real values 14.3.8
    of a based literal 2.4.1, 2.4.2
    of a decimal literal 2.4.1

Exponentiating operator 4.5; 4.5.6
  [see also: highest precedence operator]
    in a factor 4.4
    overloaded 6.7

Exponentiation compound delimiter 2.2
  [see also: double star compound delimiter]

Exponentiation operation 4.5.6

Export pragmas 13.9a

EXPORT_EXCEPTION (DEC Ada predefined pragma) **13.9a.3.2**; 13.9a, 13.9a.3, B

EXPORT_FUNCTION (DEC Ada predefined pragma) **13.9a.1.3**; 13.9a, 13.9a.1, B

EXPORT_OBJECT (DEC Ada predefined pragma) **13.9a.2.2**; 13.9a, 13.9a.2, B

EXPORT_PROCEDURE (DEC Ada predefined pragma) **13.9a.1.3**; 13.9a, 13.9a.1, B

EXPORT_VALUED_PROCEDURE (DEC Ada predefined pragma) **13.9a.1.3**; 13.9a, 13.9a.1, B

Expression 4.4; D
  [see also: compile time evaluation, default expression, delay expression, evaluation, qualified expression, simple expression, static expression, universal type expression]
    as an actual parameter 6.4, 6.4.1
    as a condition 5.3
    as a generic actual parameter 12.3; 12.3.1
    as the argument of a pragma 2.8

in an actual parameter of a
conditional entry call 9.7.2
in an actual parameter of an
entry call statement 9.5
in an actual parameter of a timed
entry call 9.7.3
in an allocator 4.8
in an assignment statement 5.2
in an attribute designator 4.1.4
in a case statement 5.4
in a choice in a case statement
5.4
in a component association 4.3
in a component declaration 3.7
in a constraint 3.3.2
in a conversion 4.6
in a discriminant association 3.7.2
in a discriminant specification
3.7.1
in a generic formal part 12.1
in an indexed component 4.1.1
in a length clause 13.2
in a name of a variable 5.2, 6.4.1
in a number declaration 3.2
in an object declaration 3.2, 3.2.1
in a parameter specification 6.1
in a primary 4.4
in a qualified expression 4.7
in a representation clause 13.1
in a return statement 5.8
in a specification of a derived
subprogram 3.4
in a type conversion 8.7
including the name of a private
type 7.4.1
specifying an entry in a family
4.1.1
specifying the value of an index
4.1.1
with a boolean result 4.5.1, 4.5.2,
4.5.6

Extended_digit in a based literal **2.4.2**

External file 14.1
[see also: file]

Factor 4.4
in a term 4.4

FALSE boolean enumeration literal
3.5.3; C

Family of entries
[see: entry family]

FETCH_FROM_ADDRESS (DEC Ada
generic function)
[see: system.fetch_from_address]

F_FLOAT (DEC Ada predefined type)
[see: system.f_float]

F_floating (DEC Ada floating point type
representation) **3.5.7**; 3.5.7a, 3.5.7b)
values of for machine-dependent
attributes F

FIELD (predefined integer subtype)
14.3.5; 14.3.7, 14.3.10

File (object of a file type) **14.1**
[see also: external file]
arrangement of values in 14.1a
mixed-type elements in 14.1a,
14.2a

File Definition Language (FDL) 14.1b

File descriptor D

File management 14.2.1
in text_io 14.3.1
in DEC Ada relative and indexed
packages 14.2a.1
in DEC Ada mixed-type packages
14.2b.1

File terminator 14.3; 14.3.1, 14.3.4,
14.3.5, 14.3.6, 14.3.7, 14.3.8, 14.3.9

FILE_MODE (input-output type)
in an instance of direct_io 14.1,
14.2.1; 14.2.5
in an instance of sequential_io
14.1, 14.2.1; 14.2.3
in text_io 14.1, 14.2.1; 14.3.10

Floating point type 3.5.7; D

>[see also: numeric type, real type,
>scalar type, system.max_digits]
>>accuracy of an operation 4.5.7
>>as a generic actual type 12.3.3
>>as a generic formal type 12.1.2
>>error bounds 4.5.7; 3.5.6
>>operation 3.5.8; 4.5.3, 4.5.4, 4.5.5,
>>4.5.6
>>result of an operation out of range
>>of the type 4.5.7
>>DEC Ada representation of 3.5.7,
>>3.5.7a, 3.5.7b

Font design of graphical symbols **2.1**

For loop

>[see: loop statement]

Forcing occurrence (of a name leading
to default determination of representation)
13.1

FORE (predefined attribute) for a fixed
point type **3.5.10**; A

Fore field of text_io input or output
14.3.8, **14.3.10**; 14.3.5

FORM (input-output function)

>>in an instance of direct_io 14.2.1;
>>14.2.5
>>in an instance of sequential_io
>>14.2.1, 14.2.3
>>in text_io 14.2.1; 14.3.10
>>raising an exception 14.4
>>in an instance of indexed_io
>>14.2.1, 14.2a.5
>>in an instance of relative_io
>>14.2.1, 14.2a.3
>>in direct_mixed_io 14.2.1, 14.2b.4
>>in indexed_mixed_io 14.2.1,
>>14.2b.10
>>in relative_mixed_io 14.2.1,
>>14.2b.8
>>in sequential_mixed_io 14.2.1,
>>14.2b.4

Form feed format effector **2.1**

Form string of a file **14.1**; 14.2.1, 14.2.3,
14.2.5, 14.3.10

>>DEC Ada interpretation of 14.1b

Formal object

>[see: generic formal object]

Formal parameter 6.1; D; (of an entry)
9.5; 3.2, 3.2.1; (of a function) 6.5; (of an
operator) 6.7; (of a subprogram) 6.1, 6.2,
6.4; 3.2, 3.2.1, 6.3

>[see also: actual parameter, default
>expression, entry, generic formal
>parameter, mode, object, subprogram]
>>as a constant 3.2.1
>>as an object 3.2
>>as a variable 3.2.1
>>names and overload resolution 6.6
>>of a derived subprogram 3.4
>>of a generic formal subprogram
>>12.1, 12.1.3
>>of a main program 10.1
>>of an operation 3.3.3
>>of a renamed entry or subprogram
>>8.5
>>whose type is an array type 3.6.1
>>whose type is a limited type 7.4.4
>>whose type is a record type 3.7.2
>>whose type is a task type 9.2
>>maximum number in a DEC Ada
>>subprogram or entry declaration
>>F

Formal part 6.1; 6.4

>[see also: generic formal part,
>parameter type profile]
>>conforming to another 6.3.1
>>in an accept statement 9.5
>>in an entry declaration 9.5
>>in a subprogram specification 6.1
>>must not include a pragma 2.8

Formal subprogram

>[see: generic formal subprogram]

Formal type

>[see: generic formal type]

Format effector 2.1
[see also: carriage return, form feed, horizontal tabulation, line feed, vertical tabulation]
    as a separator 2.2
    in an end of line 2.2

Format of text_io input or output **14.3.5**, **14.3.7**, **14.3.8**, **14.3.9**

Formula
[see: expression]

Frame 11.2
    and optimization 11.6
    in which an exception is raised 11.4.1, 11.4.2

Full declaration
    of a deferred constant 7.4.3

Full type declaration 3.3.1
    discriminant part is not elaborated 3.3.1
    of an incomplete type 3.8.1
    of a limited private type 7.4.4
    of a private type 7.4.1; 7.4.2

Function 6.1, **6.5**; 6, 12.3, D
[see also: operator, parameter and result type profile, parameter, predefined function, result subtype, return statement, subprogram]
    as a main program 10.1
    renamed 8.5
    result [see: returned value]
    that is an attribute 4.1.4; 12.3.6
    exporting 13.9a.1.4
    importing 13.9a.1.1

Function body
[see: subprogram body]

Function call 6.4; 6
[see also: actual parameter, subprogram call]
    as a prefix 4.1, 4.1.3
    as a primary 4.4
    in a static expression 4.9

    with a parameter of a derived type 3.4
    with a result of a derived type 3.4

Function specification
[see: subprogram specification]

Garbage collection 4.8

Generic actual object 12.3.1; 12.1.1
[see also: generic actual parameter]

Generic actual parameter 12.3; 12
[see also: generic actual object, generic actual subprogram, generic actual type, generic association, generic formal parameter, generic instantiation, matching]
    cannot be a universal_fixed operation 4.5.5
    for a generic formal access type 12.3.5
    for a generic formal array type 12.3.4
    for a generic formal object 12.1.1
    for a generic formal private type 12.3.2
    for a generic formal scalar type 12.3.3
    for a generic formal subprogram 12.1.3; 12.3.6
    for a generic formal type 12.1.2
    is not static 4.9
    that is an array aggregate 4.3.2
    that is a loop parameter 5.5
    that is a task type 9.2

Generic actual part 12.3

Generic actual subprogram 12.1.3, **12.3.6**
[see also: generic actual parameter]

Generic actual type
[see: generic actual parameter]
for a generic formal access type
12.3.5
for a generic formal array type
12.3.4
for a generic formal scalar type
12.3.3
for a generic formal type with
discriminants 12.3.2
for a generic private formal type
12.3.2
that is a private type 7.4.1

Generic association 12.3
[see also: generic actual parameter,
generic formal parameter]
named generic association 12.3
named generic association for
selective visibility 8.3
positional generic association 12.3

Generic body 12.2; 12.1, 12.1.2, 12.3.2
[see also: body stub, elaboration
of . . .]
in a package body 7.1
including an exception handler
11.2; 11
including an exit statement 5.7
including a goto statement 5.9
including an implicit declaration
5.1
must be in the same declarative
region as the declaration 3.9, 7.1
not yet elaborated at an
instantiation 3.9
inlined for each instantiation
12.1a

Generic declaration 12.1; 12, 12.1.2,
12.2
[see also: elaboration of . . .]
and body as a declarative region
8.1
and proper body in the same
compilation 10.3
as a basic declaration 3.1
as a later declarative item 3.9

as a library unit 10.1
in a package specification 7.1
recompiled 10.3

Generic formal object 12.1, **12.1.1**; 3.2,
12.3, 12.3.1
[see also: default expression, generic
formal parameter]
of an array type 3.6.1
of a record type 3.7.2

Generic formal parameter 12.1, **12.3**;
12, D
[see also: generic actual parameter,
generic association, generic formal
object, generic formal subprogram,
generic formal type, matching, object]
as a constant 3.2.1
as a variable 3.2.1
of a limited type 7.4.4
of a task type 9.2

Generic formal part 12.1; 12, D

Generic formal subprogram 12.1,
12.1.3; 12.1.2, 12.3, 12.3.6
[see also: generic formal parameter]
formal function 12.1.3
with the same name as another
12.3

Generic formal type 12.1, **12.1.2**; 12.3
[see also: constraint on . . . ,
discriminant of . . . , generic formal
parameter, subtype indication . . .]
as index or component type of a
generic formal array type 12.3.4
formal access type 12.1.2, 12.3.5
formal array type 12.1.2, 12.3.4
formal array type (constrained)
12.1.2
formal discrete type 12.1.2
formal enumeration type 12.1.2
formal fixed point type 12.1.2
formal floating point type 12.1.2
formal integer type 12.1.2
formal limited private type 12.3.2
formal limited type 12.1.2
formal part 12.1.2

Hiding (of a declaration) **8.3**
    [see also: visibility]
        and renaming 8.5
        and use clauses 8.4
        due to an implicit declaration 5.1
        of a generic unit 12.1
        of a library unit 10.1
        of a subprogram 6.6
        of or by a derived subprogram 3.4
        of the package standard 10.1
        within a subunit 10.2

Highest precedence operator 4.5
    [see also: abs, arithmetic operator,
    exponentiating operator, not unary
    operator, overloading of an operator,
    predefined operator]
        as an operation of a discrete type
        3.5.5
        as an operation of a fixed point
        type 3.5.10
        as an operation of a floating point
        type 3.5.8
        overloaded 6.7

Homograph (declaration) **8.3**
    [see also: overloading]
        and use clauses 8.4

Horizontal tabulation
        as a separator 2.2
        character in a comment 2.7
        format effector 2.1
        in text_io input 14.3.5

Hyphen character 2.1
    [see also: minus character]
        starting a comment 2.7

IDENT (DEC Ada predefined pragma) **B**

Identifier 2.3; 2.2
    [see also: direct visibility, loop
parameter, name, overloading of . . . ,
scope of . . . , simple name, visibility]
        and an adjacent separator 2.2
        as an attribute designator 4.1.4
        as a designator 6.1
        as a reserved word 2.9
        as a simple name 4.1
        can be written in the basic
        character set 2.10
        denoting an object 3.2.1
        denoting a value 3.2.2
        in a deferred constant declaration
        7.4.3
        in an entry declaration 9.5
        in an exception declaration 11.1
        in a generic instantiation 12.3
        in an incomplete type declaration
        3.8.1
        in a number declaration 3.2.2
        in an object declaration 3.2
        in a package specification 7.1
        in a private type declaration 7.4;
        7.4.1
        in a renaming declaration 8.5
        in a subprogram specification 6.1
        in a task specification 9.1
        in a type declaration 3.3.1; 7.4.1
        in its own declaration 8.3
        in pragma system_name 13.7
        of an argument of a pragma 2.8
        of an enumeration value 3.5.1
        of a formal parameter of a generic
        formal subprogram 12.1.3
        of a generic formal object 12.1,
        12.1.1
        of a generic formal subprogram
        12.1; 12.1.3
        of a generic formal type 12.1;
        12.1.2
        of a generic unit 12.1
        of a library unit 10.1
        of a pragma 2.8
        of a subprogram 6.1
        of a subtype 3.3.2
        of a subunit 10.2

of homograph declarations 8.3
overloaded 6.6
versus simple name 3.1
maximum DEC Ada length F

Identifier list 3.2
in a component declaration 3.7
in a deferred constant declaration 7.4
in a discriminant specification 3.7.1
in a generic parameter declaration for generic formal objects 12.1
in a number declaration 3.2
in an object declaration 3.2
in a parameter specification 6.1

Identity operation 4.5.4

IEEE double float (DEC Ada floating point type representation) **3.5.7**
values of for machine-dependent attributes F

IEEE_DOUBLE_FLOAT (DEC Ada predefined type)
[see: system.ieee_double_float]

IEEE single float (DEC Ada floating point type representation) **3.5.7**
values of for machine-dependent attributes F

IEEE_SINGLE_FLOAT (DEC Ada predefined type)
[see: system.ieee_single_float]

If statement 5.3
[see also: statement]
as a compound statement 5.1

Illegal 1.6

IMAGE (predefined attribute) **3.5.5**; A

Immediate scope 8.2; 8.3

Immediately within (a declarative region)
[see: occur immediately within]

Implementation defined
[see: system dependent]

Implementation defined pragma F

Implementation dependent
[see: system dependent]

Implicit conversion 4.6
[see also: conversion operation, explicit conversion, subtype conversion]
of an integer literal to an integer type 3.5.4
of a real literal to a real type 3.5.6
of a universal expression 3.5.4, 3.5.6
of a universal real expression 4.5.7

Implicit declaration 3.1; 4.1
[see also: scope of . . .]
by a type declaration 4.5
hidden by an explicit declaration 8.3
of a basic operation 3.1, 3.3.3
of a block name, loop name, or label 5.1; 3.1
of a derived subprogram 3.3.3, 3.4
of an enumeration literal 3.3.3
of an equality operator 6.7
of an exception due to an instantiation 11.1
of a library unit 8.6, 10.1
of a predefined operator 4.5
of universal_fixed operators 4.5.5

Implicit initialization of an object
[see: allocator, default initial value]

Implicit representation clause
for a derived type 3.4

Import pragmas 13.9

IMPORT_ADDRESS (DEC Ada predefined function)
[see:system.import_address]

IMPORT_EXCEPTION (DEC Ada predefined pragma) **13.9a.3.1**; 13.9a, 13.9a.3, B

IMPORT_FUNCTION (DEC Ada predefined pragma) **13.9a.1.1**; 13.9a, 13.9a.1, B

IMPORT_LARGEST_INTEGER (DEC Ada predefined function)
[see:system.import_largest_integer]

IMPORT_OBJECT (DEC Ada predefined pragma) **13.9a.2.1**; 13.9a, 13.9a.2, 13.9a.2.3, B

IMPORT_PROCEDURE (DEC Ada predefined pragma) **13.9a.1.1**; 13.9a, 13.9a.1, B

IMPORT_VALUE (DEC Ada predefined function)
[see: system.import_value]

IMPORT_VALUED_PROCEDURE (DEC Ada predefined pragma) **13.9a.1.1**; 13.9a, 13.9a.1, B

In membership test
[see: membership test]

In mode
[see: mode in]

In out mode
[see: mode in out]

IN_FILE (input-output file mode enumeration literal) **14.1**

Inclusive disjunction
[see: logical operator]

Incompatibility (of constraints)
[see: compatibility]

Incomplete type 3.8.1
corresponding full type declaration 3.3.1

Incomplete type declaration 3.8.1; 3.3.1, 7.4.1
as a portion of a declarative region 8.1

Incorrect order dependence 1.6
[see also: program error]
assignment statement 5.2
bounds of a range constraint 3.5
component association of an array aggregate 4.3.2
component association of a record aggregate 4.3.1
component subtype indication 3.6
default expression for a component 3.2.1
default expression for a discriminant 3.2.1
expression 4.5
index constraint 3.6
library unit 10.5
parameter association 6.4
prefix and discrete range of a slice 4.1.2

Index 3.6; D
[see also: array, discrete type, entry index]

INDEX (input-output function)
in an instance of direct_io 14.2.4; 14.2.5
in an instance of relative_io 14.2a.2, 14.2a.3
in direct_mixed_io 14.2b.5, 14.2b.6
in relative_mixed_io 14.2b.7, 14.2b.8

Index constraint 3.6, **3.6.1**; D
[see also: dependence on a discriminant]
ignored due to index_check suppression 11.7
in an allocator 4.8
in a constrained array definition 3.6
in a subtype indication 3.3.2

on an access type 3.8
violated 11.1

Index of an element in a direct access file **14.2**; 14.2.4

Index in a relative access file **14.2a**

Index range 3.6
matching 4.5.2

Index subtype 3.6

Index subtype definition 3.6

Index type
of a choice in an array aggregate 4.3.2
of a generic formal array type 12.3.4

Index_check
[see: constraint_error, suppress]
[see also: address (DEC Ada predefined attribute), size (DEC Ada predefined attribute), bit (DEC Ada predefined attribute)]

Indexed access file 14.2a

Indexed component 4.1.1; 3.6, D
as a basic operation 3.3.3; 3.3, 3.6.2, 3.8.2
as a name 4.1
as the name of an entry 9.5
of a value of a generic formal array type 12.1.2

INDEXED_IO (DEC Ada predefined input-output package) **14.2a.4**; 14.2a, 14.2a.5
exceptions 14.4; 14.5, 14.5a
element locking in 14.2a
requisite specification of FORM parameter with 14.1b, 14.2a.1
specification 14.2a.5

INDEXED_MIXED_IO (DEC Ada predefined input-output package) **14.2b.9**; 14.2b.10]
exceptions 14.4; 14.5, 14.5a
requisite specification of FORM parameter with 14.1b
specification 14.2b.10

Indication
[see: subtype indication]

Indivisible access of a shared variable **9.11**

Inequality compound delimiter 2.2

Inequality operator 4.5; 4.5.2
[see also: limited type, relational operator]
cannot be explicitly declared 6.7
for an access type 3.8.2
for an array type 3.6.2
for a generic formal type 12.1.2
for a real type 4.5.7
for a record type 3.7.4
not available for a limited type 7.4.4

Initial value (of an object) **3.2.1**
[see also: allocator, composite type, default expression, default initial value, default initialization]
in an allocator 4.8; 3.8, 7.4.4
of an array object 3.6.1
of a constant 3.2.1
of a constant in a static expression 4.9
of a discriminant of a formal parameter 6.2
of a discriminant of an object 3.7.2
of a limited private type object 7.4.4
of an object declared in a package 7.1
of an out mode formal parameter 6.2
of a record object 3.7.2

Initialization
[see: assignment, default expression, default initialization, initial value]

INLINE (predefined pragma) **6.3.2**; B
creating recompilation dependence 10.3

INLINE_GENERIC (DEC Ada predefined pragma) **12.1a**; B
creating recompilation dependence 10.3

INOUT_FILE (input-output file_mode enumeration literal) **14.1**

Input-output 14
[see also: direct_io, io_exceptions, low_level_io, sequential_io, text_io]
[see also: aux_io_exceptions, direct_mixed_io, indexed_io, indexed_mixed_io, relative_io, relative_mixed_io, sequential_mixed_io]
at device level 14.6
exceptions 14.4; 14.5
with a direct access file 14.2.4
with a sequential file 14.2.2
with a text file 14.3
auxiliary exceptions 14.5a
implementation-dependent characteristics of F
with an indexed file 14.2a
with a mixed-type file 14.2b
with a relative file 14.2a

INSQHI (DEC Ada procedure)
[see: system.insqhi]

INSQ_STATUS (DEC Ada type)
[see: system.insq_status]

INSQTI (DEC Ada procedure)
[see: system.insqti]

Instance
[see: generic instance]

Instantiation
[see: generic instantiation]

INTEGER (predefined type) **3.5.4**; C
as base type of a loop parameter 5.5
as default type for the bounds of a discrete range 3.6.1; 9.5

Integer literal 2.4
[see also: based integer literal, universal_integer type]
as a bound of a discrete range 9.5
as a universal_integer literal 3.5.4
in based notation 2.4.2
in decimal notation 2.4.1

Integer part
as a base of a based literal 2.4.2
of a decimal literal 2.4.1

Integer predefined type 3.5.4
[see also: INTEGER, LONG_INTEGER, SHORT_INTEGER]

[see also: SHORT_SHORT_INTEGER, system.integer]

Integer subtype
[see: priority]
due to an integer type definition 3.5.4

Integer type 3.5.4; 3.3, 3.5, D
[see also: discrete type, numeric type, predefined type, scalar type, system.maxint, system.min_int, universal_integer type]
as a generic formal type 12.1.2
as a generic parameter 12.3.3
operation 3.5.5; 4.5.3, 4.5.4, 4.5.5, 4.5.6
result of a conversion from a numeric type 4.6
result of an operation out of range of the type 4.5

Integer type declaration
[see: integer type definition]

Integer type definition 3.5.4; 3.3.1
[see also: elaboration of . . .]

Integer type expression
in a length clause 13.2
in a record representation clause
13.4

INTEGER_IO (text_io inner generic
package) **14.3.6**; 14.3.10

INTERFACE (predefined pragma) **13.9**; B

with imported subprograms
13.9a.1.1

Interface to other languages 13.9

INTERFACE_NAME (DEC Ada
predefined pragma) **13.9b**; B

Interrupt 13.5

Interrupt entry 13.5.1
[see also: address attribute]
[see also: ast_entry]

Interrupt queue
[see: entry queue]

IO_EXCEPTIONS (predefined input-
output package) **14.4**; 14, 14.1, 14.2.3,
14.2.5, 14.3.10, C
specification 14.5

IS_OPEN (input-output function)
in an instance of direct_io 14.2.1;
14.2.5
in an instance of sequential_io
14.2.1, 14.2.3
in text_io 14.2.1; 14.3.10
in an instance of indexed_io
14.2.1, 14.2a.5
in an instance of relative_io
14.2.1, 14.2a.3
in direct_mixed_io 14.2.1, 14.2b.6
in indexed_mixed_io 14.2.1,
14.2b.10
in relative_mixed_io 14.2.1,
14.2b.8

in sequential_mixed_io 14.2.1,
14.2b.4

ISO (international organization for
standardization) **2.1**

ISO seven bit coded character set **2.1**

Item
[see: basic declarative item, later
declarative item]

Iteration scheme 5.5
[see also: discrete type]

Key in an indexed access file **14.2a**
exact and inexact matching of
14.2a

KEY_ERROR (DEC Ada input-output
exception) 14.4; 14.2a.4, 14.2a.5, 14.2b.9,
14.2b.10, 14.5a

Label 5.1
[see also: address attribute, name,
statement]
declaration 5.1
implicitly declared 3.1
target of a goto statement 5.9

Label bracket
compound delimiter 2.2

Labeled statement 5.1
in a code statement 13.8

Language interpretations G

followed by an inline pragma
6.3.2
included in the predefined
package standard 8.6
must not be subject to an address
clause 13.5
named in a use clause 10.5
named in a with clause 10.1.1;
10.3, 10.5
recompiled 10.3
scope 8.2
subject to an interface pragma
13.9
that is a package 7.1
visibility due to a with clause 8.3
whose name is needed in a
compilation unit 10.1.1
with a body stub 10.2
maximum number in DEC Ada F

Limited private type 7.4.4
[see also: private type]
as a generic actual type 12.3.2
as a generic formal type 12.1.2

Limited type 7.4.4; 9.2, 12.3.1, D
[see also: assignment, equality
operator, inequality operator,
predefined operator, task type]
as a full type 7.4.1
component of a record 3.7
generic formal object 12.1.1
in an object declaration 3.2.1
limited record type 3.7.4
operation 7.4.4; 4.5.2
parameters for explicitly declared
equality operators 6.7

Line 14.3, **14.3.4**

LINE (text_io function) **14.3.4**; 14.3.10
raising an exception 14.4

Line feed format effector **2.1**

Line length 14.3, **14.3.3**; 14.3.1, 14.3.4,
14.3.5, 14.3.6

Line terminator 14.3; 14.3.4, 14.3.5,
14.3.6, 14.3.7, 14.3.8, 14.3.9

LINE_LENGTH (text_io function) **14.3.3**,
14.3.4; 14.3.3, 14.3.10
raising an exception 14.4

Linker D

List
[see: component list, identifier_list]

LIST (predefined pragma) **B**

Listing of program text
[see: list pragma, page pragma]

Literal 4.2; D
[see also: based literal, character
literal, decimal literal, enumeration
literal, integer literal, null literal,
numeric literal, overloading of . . . ,
real literal, string literal]
as a basic operation 3.3.3
of a derived type 3.4
of universal_integer type 3.5.4
of universal_real type 3.5.6
specification [see: enumeration
literal specification]

Local declaration 8.1
in a generic unit 12.3

LOCK_ERROR (DEC Ada input-output
exception) **14.4**; 14.2a.2, 14.2a.3, 14.2a.4,
14.2a.5, 14.2b.7, 14.2b.8, 14.2b.9, 14.2b.10,
14.5a

Logical negation operation 4.5.6

Logical operation 4.5.1

Logical operator 4.5; 4.4, 4.5.1, C
[see also: overloading of an operator,
predefined operator]
as an operation of boolean type
3.5.5
for an array type 3.6.2
in an expression 4.4
overloaded 6.7

termination 9.4
DEC Ada definition of 10.1, F
stack and stack storage for 13.2b

MAIN_STORAGE (DEC Ada predefined pragma) **13.2b**; B

MANTISSA (predefined attribute) **3.5.8**, **3.5.10**; A
DEC Ada floating point values of F

Mantissa
of a fixed point number 3.5.9
of a floating point number 3.5.7; 13.7.3

Mark
[see: type_mark]

Master (task) **9.4**

Matching components
of arrays 4.5.2; 4.5.1, 5.2.1
of records 4.5.2

Matching generic formal
and actual parameters 12.3
access type 12.3.5
array type 12.3.4
default subprogram 12.3.6; 12.1.3
object 12.3.1; 12.1.1
private type 12.3.2
scalar type 12.3.3
subprogram 12.3.6; 12.1.3
type 12.3.2, 12.3.3, 12.3.4, 12.3.5; 12.1.2

Mathematically correct result of a numeric operation **4.5**; 4.5.7

MAX_DIGITS
[see: system.max_digits]

MAX_ELEMENT_SIZE (DEC Ada mixed-type input-output function)
in direct_mixed_io 14.2b.2, 14.2b.6
in indexed_mixed_io 14.2b.2, 14.2b.10

in relative_mixed_io 14.2b.2, 14.2b.8
in sequential_mixed_io 14.2b.2, 14.2b.4

MAX_INT
[see: system.max_int]

MAX_MANTISSA
[see: system.max_mantissa]

Maximum line length 14.3

Maximum page length 14.3

Membership test 4.4, **4.5.2**
cannot be overloaded 6.7

Membership test operation 4.5
[see also: overloading of . . .]
as a basic operation 3.3.3; 3.3, 3.5.5, 3.5.8, 3.5.10, 3.6.2, 3.7.4, 3.8.2, 7.4.2
for a real type 4.5.7

MEMORY_SIZE (predefined named number)
[see: system.memory_size]

MEMORY_SIZE (predefined pragma) **13.7**; B

MFPR (DEC Ada predefined function)
[see: system.mfpr]

MIN_INT
[see: system.min_int]

Minimization of storage
[see: pack predefined pragma]

Minus
character [see: hyphen character]
character in an exponent of a numeric literal 2.4.1
delimiter 2.2
operator [see: binary adding operator, unary adding operator]
unary operation 4.5.4

Mod operator 4.5.5
[see also: multiplying operator]

MODE (input-output function)
in an instance of direct_io 14.2.1;
14.2.5
in an instance of sequential_io
14.2.1; 14.2.3
in text_io 14.2.1; 14.3.3, 14.3.4,
14.3.10
in an instance of indexed_io
14.2.1, 14.2a.5
in an instance of relative_io
14.2.1, 14.2a.3
in direct_mixed_io 14.2.1, 14.2b.6
in indexed_mixed_io 14.2.1,
14.2b.10
in relative_mixed_io 14.2.1,
14.2b.8
in sequential_mixed_io 14.2.1,
14.2b.4

Mode (of a file) **14.1**; 14.2.1
of a direct access file 14.2; 14.2.5
of a sequential access file 14.2;
14.2.3
of a text_io file 14.3.1; 14.3.4
of an indexed access file 14.2a
of a relative access file 14.2a

Mode (of a formal parameter) **6.2**; 6.1, D
[see also: formal parameter, generic
formal parameter]
of a formal parameter of a derived
subprogram 3.4
of a formal parameter of a
renamed entry or subprogram
8.5
of a generic formal object 12.1.1

Mode in for a formal parameter **6.1**, **6.2**;
3.2.1
of a function 6.5
of an interrupt entry 13.5.1

Mode in for a generic formal object
12.1.1; 3.2.1, 12.3, 12.3.1

Mode in out for a formal parameter **6.1**,
6.2; 3.2.1
of a function is not allowed 6.5
of an interrupt entry is not
allowed 13.5.1

Mode in out for a generic formal object
12.1.1; 3.2.1, 12.3, 12.3.1

Mode out for a formal parameter **6.1**, **6.2**
of a function is not allowed 6.5
of an interrupt entry is not
allowed 13.5.1

MODE_ERROR (input-output exception)
14.4; 14.2.2, 14.2.3, 14.2.4, 14.2.5, 14.3.1,
14.3.2, 14.3.3, 14.3.4, 14.3.5, 14.3.10, 14.5,
14.2a.2, 14.2a.3, 14.2a.4, 14.2a.5, 14.2b.2,
14.2b.3, 14.2b.4, 14.2b.5, 14.2b.6, 14.2b.7,
14.2b.8, 14.2b.9, 14.2b.10

Model interval of a subtype **4.5.7**

Model number (of a real type) **3.5.6**; D
[see also: real type, safe number]
accuracy of a real operation 4.5.7
of a fixed point type 3.5.9; 3.5.10
of a floating point type 3.5.7; 3.5.8

Modulus operation 4.5.5

MONTH (predefined function) **9.6**

MTPR (DEC Ada predefined procedure)
[see: system.mtpr]

Multidimensional array 3.6

Multiple
component declaration 3.7; 3.2
deferred constant declaration 7.4;
3.2
discriminant specification 3.7.1;
3.2
generic parameter declaration
12.1; 3.2
number declaration 3.2.2; 3.2
object declaration 3.2
parameter specification 6.1; 3.2

Multiplication operation 4.5.5
        accuracy for a real type 4.5.7

Multiplying operator 4.5; 4.5.5, C
    [see also: arithmetic operator,
    overloading of an operator]
        in a term 4.4
        overloaded 6.7

Must (legality requirement) **1.6**

Mutually recursive types 3.8.1; 3.3.1

NAME (input-output function)
        in an instance of direct_io 14.2.1
        in an instance of sequential_io
        14.2.1
        in text_io 14.2.1
        in an instance of indexed_io
        14.2.1, 14.2a.5
        in an instance of relative_io
        14.2.1, 14.2a.3
        in direct_mixed_io 14.2.1, 14.2b.8
        in indexed_mixed_io 14.2.1,
        14.2b.10
        in relative_mixed_io 14.2.1,
        14.2b.6
        in sequential_mixed_io 14.2.1,
        14.2b.4

NAME (predefined type)
    [see: system.name]

Name (of an entity) **4.1**; 2.3, 3.1, D
    [see also: attribute, block name,
    denote, designator, evaluation of . . . ,
    forcing occurrence, function call,
    identifier, indexed component, label,
    loop name, loop parameter, operator
    symbol, renaming declaration,
    selected component, simple name,
    slice, type_mark, visibility]
        as a prefix 4.1
        as a primary 4.4

        as the argument of a pragma 2.8
        as the expression in a case
        statement 5.4
        conflicts 8.5
        declared by renaming is not
        allowed as prefix of certain
        expanded names 4.1.3
        declared in a generic unit 12.3
        denoting an entity 4.1
        denoting an object designated by
        an access value 4.1
        generated by an implementation
        13.4
        starting with a prefix 4.1; 4.1.1,
        4.1.2, 4.1.3, 4.1.4

Name string (of a file) **14.1**; 14.2.1,
14.2.3, 14.2.5, 14.3, 14.3.10, 14.4

NAME_ERROR (input-output exception)
14.4; 14.2.1, 14.2.3, 14.2.5, 14.3.10, 14.5,
14.2a.3, 14.2a.5, 14.2b.6, 14.2b.8, 14.2b.10

Named association 6.4.2, **D**
    [see also: component association,
    discriminant association, generic
    association, parameter association]

Named block statement
    [see: block name]

Named loop statement
    [see: loop name]

Named number 3.2; 3.2.2
        as an entity 3.1
        as a primary 4.4
        in a static expression 4.9

NATURAL (predefined integer subtype) **C**

Negation
    [see: logical negation operation]

Negation operation (numeric) **4.5.4**

Negative exponent
        in a numeric literal 2.4.1
        to an exponentiation operator
        4.5.6

NEW_LINE (text_io procedure) **14.3.4**; 14.3.5, 14.3.6, 14.3.10
> raising an exception 14.4

NEW_PAGE (text_io procedure) **14.3.4**; 14.3.10
> raising an exception 14.4

Next element in an indexed access file **14.2a**

No other effect
> [see: elaboration has no other effect]

NO_ADDR (DEC Ada predefined constant)
> [see:system.no_addr]

NO_AST_HANDLER (DEC Ada predefined constant) **9.12a**; F
> [see: system.no_ast_handler]

NON_ADA_ERROR (DEC Ada predefined exception)
> [see: system.non_ada_error]

Noncontiguous array D; 13.9a.1.2

Not equal
> compound delimiter [see: inequality compound delimiter]
> operator [see: relational operator]

Not in membership test
> [see: membership test]

Not unary operator
> [see: highest precedence operator]
> as an operation of an array type 3.6.2
> as an operation of boolean type 3.5.5
> in a factor 4.4

Not yet elaborated 3.9

Null access value 3.8; 3.4, 4.2, 6.2, 11.1
> [see also: default initial value of an access type object]
> causing constraint_error 4.1
> not causing constraint_error 11.7

Null array 3.6.1; 3.6
> aggregate 4.3.2
> and relational operation 4.5.2
> as an operand of a catenation 4.5.3

Null component list 3.7

Null literal 3.8, **4.2**
> [see also: overloading of . . .]
> as a basic operation 3.3.3; 3.8.2
> as a primary 4.4
> must not be the argument of a conversion 4.6

Null range 3.5
> as a choice of a variant part 3.7.3
> for a loop parameter 5.5

Null record 3.7
> and relational operation 4.5.2

Null slice 4.1.2
> [see also: array type]

Null statement 5.1
> [see also: statement]
> as a simple statement 5.1

Null string literal 2.6

NULL_ADDRESS (DEC Ada predefined constant)
> [see:system.null_address]

NULL_PARAMETER (DEC Ada predefined attribute) **13.9a.1.2**; A

Number
> [see: based literal, decimal literal]

Number declaration 3.2, **3.2.2**
> as a basic declaration 3.1

NUMBER_BASE (predefined integer subtype) **14.3.7**; 14.3.10

Numeric literal 2.4, **4.2**; 2.2, 2.4.1, 2.4.2
> [see also: universal type expression]
> and an adjacent separator 2.2
> as a basic operation 3.3.3
> as a primary 4.4

as the parameter of value
attribute 3.5.5
as the result of image attribute
3.5.5
assigned 5.2
can be written in the basic
character set 2.10
in a conforming construct 6.3.1
in a static expression 4.9
in pragma memory_size 13.7
in pragma storage_unit 13.7

Numeric operation of a universal type
4.10

Numeric type 3.5
[see also: conversion, fixed point type,
floating point type, integer type, real
type, scalar type]
operation 4.5, 4.5.2, 4.5.3, 4.5.4,
4.5.5, 4.5.6

Numeric type expression
in a length clause 13.2

Numeric value of a named number **3.2**

NUMERIC_ERROR (predefined
exception) **11.1**
[see also: suppress pragma]
not raised due to lost overflow
conditions 13.7.3
not raised due to optimization
11.6
raised by a numeric operator 4.5
raised by a predefined integer
operation 3.5.4
raised by a real result out of
range of the safe numbers 4.5.7
raised by a universal expression
4.10
raised by integer division
remainder or modulus 4.5.5
raised due to a conversion out of
range 3.5.4, 3.5.6

Object 3.2; 3.2.1, D
[see also: address attribute, allocator,
collection, component, constant,
formal parameter, generic formal
parameter, initial value, loop
parameter, size attribute, storage bits
allocated, subcomponent, variable]
as an actual parameter 6.2
as a generic formal parameter
12.1.1
created by an allocator 4.8
created by elaboration of an object
declaration 3.2.1
of an access type [see: access type
object]
of a file type [see: file]
of a task type [see: task object]
renamed 8.5
subject to an address clause 13.5
subject to a representation clause
13.1
subject to a suppress pragma 11.7

exporting to non-Ada programs
13.9a.2, 13.9a.2.2, 13.9a.2.3
importing from non-Ada programs
13.9a.2, 13.9a.2.1, 13.9a.2.3
maximum number of bits in DEC
Ada F
maximum number declared with
PSECT_OBJECT pragma F

Object declaration 3.2, **3.2.1**
[see also: elaboration of . . . , generic
parameter declaration]
as a basic declaration 3.1
as a full declaration 7.4.3
implied by a task declaration 9.1
in a package specification 7.1
of an array object 3.6.1
of a record object 3.7.2
with a limited type 7.4.4
with a task type 9.2; 9.3

Object designated by an access value
3.2, 3.8, 4.8; 4.1.3, 5.2, 9.2, 11.1)
[see also: task object designated . . .]
by an access value denoted by a
name 4.1
by an access-to-array type 3.6.1
by an access-to-record type 3.7.2
by a generic formal access type
value 12.3.5

Object file D

Object module
for a subprogram written in
another language 13.9
OpenVMS D

Obsolete compilation unit (due to
recompilation) **10.3**

Occur immediately within (a
declarative region) **8.1**; 8.3, 8.4, 10.2

Omitted parameter association for a
subprogram call **6.4.2**

OPEN (input-output procedure)
in an instance of direct_io 14.2.1;
14.1, 14.2.5
in an instance of sequential_io
14.2.1; 14.1, 14.2.3
in text_io 14.2.1; 14.1, 14.3.1,
14.3.10
raising an exception 14.4
in an instance of indexed_io
14.2.1, 14.2a.5
in an instance of relative_io
14.2.1, 14.2a.3
in direct_mixed_io 14.2.1, 14.2b.6
in indexed_mixed_io 14.2.1,
14.2b.10
in relative_mixed_io 14.2.1,
14.2b.8
in sequential_mixed_io 14.2.1,
14.2b.4

Open alternative 9.7.1
[see also: alternative]
accepting a conditional entry call
9.7.2
accepting a timed entry call 9.7.3

Open file 14.1

Operation 3.3, **3.3.3**; D
[see also: basic operation, direct
visibility, operator, predefined
operation, visibility by selection,
visibility]
classification 3.3.3
of an access type 3.8.2
of an array type 3.6.2
of a discrete type 3.5.5
of a fixed point type 3.5.10
of a floating point type 3.5.8
of a generic actual type 12.1.2
of a generic formal type 12.1.2;
12.3
of a limited type 7.4.4
of a private type 7.4.2; 7.4.1
of a record type 3.7.4
of a subtype 3.3
of a subtype of a discrete type
3.5.5
of a type 3.3
of a universal type 4.10
propagating an exception 11.6
subject to a suppress pragma 11.7

Operator 4.5; 4.4, C, D
[see also: binary adding operator,
designator, exponentiating operator,
function, highest precedence operator,
logical operator, multiplying operator,
overloading of . . . , predefined
operator, relational operator, unary
adding operator]
as an operation 3.3.3 [see also:
operation]
implicitly declared 3.3.3
in an expression 4.4
in a static expression 4.9
of a derived type 3.4
of a generic actual type 12.1.2

overloaded 6.7; 6.6

renamed 8.5

Operator declaration 6.1; 4.5, 6.7

Operator symbol 6.1

[see also: direct visibility, overloading of . . . , scope of . . . , visibility by selection, visibility]

as a designator 6.1

as a designator in a function declaration 4.5

as a name 4.1

before arrow compound delimiter 8.3

declared 3.1

declared in a generic unit 12.3

in a renaming declaration 8.5

in a selector 4.1.3

in a static expression 4.9

not allowed as the designator of a library unit 10.1

of a generic formal function 12.1.3, 12.3

of homograph declarations 8.3

overloaded 6.7; 6.6

Optimization 10.6

[see also: optimize pragma]

and exceptions 11.6

OPTIMIZE (predefined pragma) **B**

Optional parameter

in imported subprogram 13.9a.1.2

Or else control form

[see: short circuit control form]

Or operator

[see: logical operator]

OR_ATOMIC (DEC Ada predefined procedure)

[see: system.or_atomic]

Order

[see: lexicographic order]

Order not defined by the language

[see: incorrect order dependence]

Order of application of operators in an expression **4.5**

Order of compilation (of compilation units) **10.1**, **10.3**; 10.1.1, 10.4

creating recompilation dependence 10.3

Order of copying back of **out** and **in out** formal parameters **6.4**

Order of elaboration 3.9

[see also: incorrect order dependence]; (of compilation units) 10.5; 10.1.1

Order of evaluation 1.6

[see also: incorrect order dependence]

and exceptions 11.6

of conditions in an if statement 5.3

of default expressions for components 3.2.1

of expressions and the name in an assignment statement 5.2

of operands in an expression 4.5

of parameter associations in a subroutine call 6.4

of the bounds of a range 3.5

of the conditions in a selective wait 9.7.1

Order of execution of statements **5.1**

[see also: incorrect order dependence]

Ordering operator 4.5; 4.5.2

Ordering relation 4.5.2

[see also: relational operator]

for a real type 4.5.7

of an enumeration type preserved by a representation clause 13.3

of a scalar type 3.5

Other effect

[see: elaboration has no other effect]

Other special character 2.1
[see also: graphic character]

Others 3.7.3
as a choice in an array aggregate
4.3.2
as a choice in a case statement
alternative 5.4
as a choice in a component
association 4.3
as a choice in a record aggregate
4.3.1
as a choice in a variant part 3.7.3
as an exception choice 11.2
as a choice for handling
OpenVMS conditions signaled
from non-Ada code 11.2

Out mode
[see: mode out]

OUT_FILE (input-output file mode
enumeration literal) **14.1**

Overflow of real operations **4.5.7**; 13.7.3

Overflow_check
[see: numeric_error, suppress]

Overlapping scopes
[see: hiding, overloading]

Overlapping slices in array assignment
5.2.1

Overlaying of objects or program units
13.5

Overloading 8.3; D
[see also: designator, homograph
declaration, identifier, operator
symbol, scope, simple name,
subprogram, visibility]
and visibility 8.3
in an assignment statement 5.2
in an expression 4.4
resolution 6.6
resolution context 8.7
resolved by explicit qualification
4.7

Overloading of
an aggregate 3.4
an allocator 4.8
a declaration 8.3
a designator 6.6; 6.7
an entry 9.5
an enumeration literal 3.5.1; 3.4
a generic formal subprogram 12.3
a generic unit 12.1
an identifier 6.6
a library unit by a locally
declared subprogram 10.1
a library unit by means of
renaming 10.1
a literal 3.4
a membership test 4.5.2
an operator 4.5, 6.7; 4.4, 6.1
an operator symbol 6.6; 6.7
a subprogram 6.6; 6.7
a subprogram subject to an
interface pragma 13.9
the expression in a case
statement 5.4

PACK (predefined pragma) **13.1**; B

Packable D

Package 7, **7.1**; D
[see also: deferred constant
declaration, library unit, predefined
package, private part, program unit,
visible part]
as a generic instance 12.3; 12
including a raise statement 11.3
named in a use clause 8.4
renamed 8.5
subject to an address clause 13.5
subject to representation clause
13.1
with a separately compiled body
10.2

of an integer value 3.5.4
of a value of a discrete type 3.5
returned by pos attribute 3.5.5

Position of a component within a record
[see: record representation clause]

Position of an element in a direct access
file **14.2**

Positional association 6.4; 6.4.2, D
[see also: component association,
discriminant association, generic
association, parameter association]

POSITIVE (predefined integer subtype)
3.6.3; 14.3.7, 14.3.8, 14.3.9, 14.3.10, C
as the index type of the string
type 3.6.3

POSITIVE_COUNT (predefined integer
subtype) **14.2.5**, **14.3.10**; 14.2.4, 14.3,
14.3.4

Potentially visible declaration 8.4

Pound sterling character 2.1

Power operator
[see: exponentiating operator]

Pragma 2.8; 2, D
[see also: predefined pragma]
applicable to the whole of a
compilation 10.1
argument that is an overloaded
subprogram name 6.3.2, 8.7, 13.9
for the specification of a
subprogram body in another
language 13.9
for the specification of program
overlays 13.5
in a code procedure body 13.8
recommending the representation
of an entity 13.1
specifying implementation
conventions for code statements
13.8

Precedence 4.5

Precision (numeric)
[see: delta, digits]

PRED (predefined attribute) **3.5.5**; 13.3, A

Predecessor
[see: pred attribute]

Predefined attribute
[see: address, base, callable,
constrained, count, first, first_bit,
image, last, last_bit, pos, pred,
range, size, small, storage_size,
succ, terminated, val, value, width]
[see: ast_entry, bit, machine_size,
null_parameter, type_class]

Predefined constant 8.6; C
[see also: system.system_name]
[see also: system.address_zero,
system.no_addr, system.null_address,
system.no_ast_handler]
for CHARACTER values [see:
ascii]

Predefined exception 8.6, **11.1**; 11.4.1,
C
[see also: constraint_error, io_
exceptions, numeric_error, program_
error, tasking_error, time_error]
[see also: aux_io_exceptions]

Predefined function 8.6; C
[see also: attribute, character literal,
enumeration literal, predefined
generic library function]

**Predefined generic library function
8.6**; C
[see also: unchecked_conversion]

**Predefined generic library package
8.6**; C
[see also: direct_io, input-output
package, sequential_io]
[see also: relative_io, indexed_io]

**Predefined generic library procedure
8.6**; C
[see also: unchecked_deallocation]

Predefined generic library subprogram 8.6; C

Predefined identifier 8.6; C

Predefined library package 8.6; C
[see also: predefined generic library package, predefined package, ascii, calendar, input-output package, io_exceptions, low_level_io, machine_code, system, text_io]
[see also: aux_io_exceptions]

Predefined library subprogram
[see: predefined generic library subprogram]

Predefined named number
[see: system.fine_delta, system.max_digits, system.max_int, system.max_mantissa, system.memory_size, system.min_int, system.storage_unit, system.tick]
[see: system.address_size]

Predefined operation 3.3, **3.3.3**; 8.6
[see also: operation, predefined operator]
accuracy for a real type 4.5.7
of a discrete type 3.5.5
of a fixed point type 3.5.10
of a floating point type 3.5.8
of a universal type 4.10
propagating an exception 11.6

Predefined operator 4.5, **8.6**; C
[see also: abs, arithmetic operator, binary adding operator, catenation, equality, exponentiating operator, highest precedence operator, inequality, limited type, logical operator, multiplying operator, operator, predefined operation, relational operator, unary adding operator]
applied to an undefined value 3.2.1
as an operation 3.3.3
for an access type 3.8.2

for an array type 3.6.2
for a record type 3.7.4
implicitly declared 3.3.3
in a static expression 4.9
of a derived type 3.4
of a fixed point type 3.5.9
of a floating point type 3.5.7
of an integer type 3.5.4
raising an exception 11.4.1

Predefined package 8.6; C
[see also: ascii, library unit, predefined library package, standard]
for input-output 14

Predefined pragma
[see: controlled, elaborate, inline, interface, list, memory_size, optimize, pack, page, priority, shared, storage_unit, suppress, system_name]
[see: ast_entry, common_object, component_alignment, export_exception, export_function, export_object, export_procedure, export_valued_procedure, float_representation, ident, import_exception, import_function, import_object, import_procedure, import_valued_procedure, inline_generic, interface_name, long_float, main_storage, psect_object, share_generic, suppress_all, task_storage, time_slice, title, volatile]

Predefined subprogram 8.6; C
[see also: input-output subprogram, library unit, predefined generic library subprogram]

Predefined subtype 8.6; C
[see also: field, natural, number_base, positive, priority]

Predefined type 8.6; C
[see also: boolean, character, count, duration, float, integer, long_float, long_integer, priority, short_float, short_integer, string, system.address,

system.name, time, universal_integer, universal_real]
[see also: long_long_float, short_short_integer, system (and individual DEC Ada types declared in system)]

Prefix 4.1; D
[see also: appropriate for a type, function call, name, selected component, selector]
    in an attribute 4.1.4
    in an indexed component 4.1.1
    in a selected component 4.1.3
    in a slice 4.1.2
    that is a function call 4.1
    that is a name 4.1

Primary 4.4
    in a factor 4.4
    in a static expression 4.9

Primary key in an indexed access file 14.2a

PRIORITY (predefined integer subtype) **9.8**; 13.7, C
[see also: task priority]

PRIORITY (predefined pragma) **9.8**; 13.7, B
[see also: task priority]

Private part (of a package) **7.2**; 7.4.1, 7.4.3, D
[see also: deferred constant declaration, private type declaration]

Private type 3.3, **7.4**, **7.4.1**; D
[see also: class of type, derived type of a private type, limited private type, type with discriminants]
    as a generic actual type 12.3.2
    as a generic formal type 12.1.2
    as a parent type 3.4
    corresponding full type declaration 3.3.1
    formal parameter 6.2
    of a deferred constant 7.4; 3.2.1
    operation 7.4.2

Private type declaration 7.4; 7.4.1, 7.4.2
[see also: private part (of a package), visible part (of a package)]
    as a generic type declaration 12.1
    as a portion of a declarative region 8.1
    including the word 'limited' 7.4.4

Procedure 6.1; 6, D
[see also: parameter and result type profile, parameter, subprogram]
    as a main program 10.1
    as a renaming of an entry 9.5
    renamed 8.5
    exporting 13.9a.1.4
    importing 13.9a.1.1

Procedure body
[see: subprogram body]
    including code statements 13.8

Procedure call 6.4; 6, D
[see also: subprogram call]

Procedure call statement 6.4
[see also: actual parameter, statement]
    as a simple statement 5.1
    with a parameter of a derived type 3.4

Procedure specification
[see: subprogram specification]

Processor 9

Profile
[see: parameter and result type profile, parameter type profile]

Program 10; D
[see also: main program]

Program legality 1.6

Program library 10.1, **10.4**; 10.5
    creation 10.4; 13.7
    manipulation and status 10.4

Qualified expression 4.7; D
        as a primary 4.4
        in an allocator 4.8
        in a case statement 5.4
        in a static expression 4.9
        qualification of an array
        aggregate 4.3.2
        to resolve an overloading
        ambiguity 6.6

Queue of entry calls
    [see: entry queue]

Queue of interrupts
    [see: entry queue]

Quotation character 2.1
        in a string literal 2.6
        replacement by percent character
        2.10

Radix of a floating point type **3.5.7**; 13.7.3

Raise statement 11.3; 11
    [see also: exception, statement]
        as a simple statement 5.1
        including the name of an
        exception 11.1

Raising of an exception **11**, **11.3**; D
    [see also: exception]
        causing a transfer of control 5.1

Range 3.5; D
    [see also: discrete range, null range]
        as a discrete range 3.6
        in a record representation clause
        13.4
        in a relation 4.4
        of an index subtype 3.6
        of an integer type containing the
        result of an operation 4.5
        of a predefined integer type 3.5.4

        of a real type containing the
        result of an operation 4.5.7
        yielded by an attribute 4.1.4

RANGE (predefined attribute) **3.6.2**;
4.1.4, A
        for an access value 3.8.2

Range constraint 3.5; D
    [see also: elaboration of . . .]
        ignored due to range_check
        suppression 11.7
        in a fixed point constraint 3.5.9
        in a floating point constraint 3.5.7
        in an integer type definition 3.5.4
        in a subtype indication 3.5; 3.3.2
        on a derived subtype 3.4
        violated 11.1

Range_check
    [see: constraint_error, suppress]
        when array parameter passing
        involves a descriptor 11.7

READ (input-output procedure)
        in an instance of direct_io 14.2.4;
        14.1, 14.2, 14.2.5
        in an instance of sequential_io
        14.2.2; 14.1, 14.2, 14.2.3
        in an instance of indexed_io
        14.2a.4, 14.2a.5
        in an instance of relative_io
        14.2a.2, 14.2a.3
        in direct_mixed_io 14.2b.5,
        14.2b.6
        in indexed_mixed_io 14.2b.9,
        14.2b.10
        in relative_mixed_io 14.2b.7,
        14.2b.8
        in sequential_mixed_io 14.2b.3,
        14.2b.4

READ_BY_KEY (DEC Ada input-output
generic procedure)
        in an instance of indexed_io
        14.2a.4, 14.2a.5
        in indexed_mixed_io 14.2b.9,
        14.2b.10

READ_EXISTING (DEC Ada input-output procedure)

        in an instance of relative_io 14.2a.2, 14.2a.3

        in relative_mixed_io 14.2b.7, 14.2b.8

Reading the value of an object **6.2**, **9.11**

READ_REGISTER (DEC Ada predefined function)

    [see system.read_register]

Real literal 2.4

    [see also: universal_real type]

        in based notation 2.4.2

        in decimal notation 2.4.1

        is of type universal_real 3.5.6

    [see also: D_floating, F_floating, G_floating, H_floating]

Real type 3.5.6; 3.3, 3.5, D

    [see also: fixed point type, floating point type, model number, numeric type, safe number, scalar type, universal_real type]

        accuracy of an operation 4.5.7

        representation attribute 13.7.3

        result of a conversion from a numeric type 4.5.7; 4.6

        result of an operation out of range of the type 4.5.7

Real type definition 3.5.6; 3.3.1, 3.5.7, 3.5.9

    [see also: elaboration of . . .]

RECEIVE_CONTROL (low_level_io procedure) **14.6**

Reciprocal operation in exponentiation by a negative integer **4.5.6**

Recompilation 10.3

        implicit 3.5.7a, 3.5.7b

Record (RMS)

    [see also: element]

        correspondence to a file element 14.1a

        correspondence to a source line 2.2

        determining number of in direct input-output file 14.2.4

        locking in a relative or indexed access file 14.2a

Record aggregate 4.3.1; 4.3

    [see also: aggregate]

        as a basic operation 3.3.3; 3.7.4

        in a code statement 13.8

Record component

    [see: component, record type, selected component]

    [see also: allocation]

Record representation clause 13.4

    [see also: first_bit attribute, last_bit attribute, position attribute]

        as a representation clause 13.1

        DEC Ada restrictions on component clause in 13.4

Record type 3.7; 3.3, D

    [see also: component, composite type, discriminant, matching components, subcomponent, type with discriminants, variant]

        formal parameter 6.2

        including a limited subcomponent 7.4.4

        operation 3.7.4

Record type declaration

    [see: record type definition, type declaration]

        as a declarative region 8.1

        determining the visibility of another declaration 8.3

Record type definition 3.7; 3.3.1

    [see also: component declaration]

Recursive
        call of a subprogram 6.1, 12.1;
        6.3.2
        generic instantiation 12.1, 12.3
        types 3.8.1; 3.3.1

Reentrant subprogram 6.1

Reference (parameter passing) **6.2**;
13.9a.1.4

Relation (in an expression) **4.4**

Relational expression
    [see: relation, relational operator]

Relational operation 4.5.2
        of a boolean type 3.5.3
        of a discrete type 3.5.5
        of a fixed point type 3.5.10
        of a floating point type 3.5.8
        of a scalar type 3.5
        result for real operands 4.5.7

Relational operator 4.5; 4.5.2, C
    [see also: equality operator,
    inequality operator, ordering relation,
    overloading of an operator, predefined
    operator]
        for an access type 3.8.2
        for an array type 3.6.2
        for a private type 7.4.2
        for a record type 3.7.4
        for time predefined type 9.6
        in a relation 4.4
        overloaded 6.7

RELATION_TYPE (DEC Ada input-output type) **14.2a**
        in an instance of indexed_io
        14.2a.4, 14.2a.5
        in indexed_mixed_io 14.2b.9,
        14.2b.10

Relative access file 14.2a

Relative address of a component within
a record
    [see: record representation clause]

RELATIVE_IO (DEC Ada predefined
input-output package) **14.2a.2**; 14.2a.3
        element locking in 14.2a
        exceptions 14.4; 14.5, 14.5a
        requisite specification of FORM
        parameter with 14.1.b, 14.2a.1
        specification 14.2a.3

RELATIVE_MIXED_IO (DEC Ada
predefined input-output package)**14.2b.7**;
14.2b.8
        exceptions 14.4; 14.5, 14.5a
        requisite specification of FORM
        parameter with 14.1b
        specification 14.2b.8

Rem operator 4.5.5
    [see also: multiplying operator]

Remainder operation 4.5.5

REMQHI (DEC Ada predefined
procedure)
    [see: system.remqhi]

REMQ_STATUS (DEC Ada predefined
type)
    [see: system.remq_status]

REMQTI (DEC Ada predefined procedure)
    [see: system.remqti]

Renaming declaration 8.5; 4.1, 12.1.3,
D
    [see also: name]
        as a basic declaration 3.1
        as a declarative region 8.1
        cannot rename a universal_fixed
        operation 4.5.5
        for an array object 3.6.1
        for an entry 9.5
        for a record object 3.7.2
        name declared is not allowed as a
        prefix of certain expanded names
        4.1.3
        to overload a library unit 10.1
        to overload a subunit 10.2
        to resolve an overloading
        ambiguity 6.6

and the pragma inline 6.3.2
and the pragma inline_generic
12.1a
and the pragma interface 13.9
for a volatile variable 9.11
for imported subprograms
13.9a.1.1

Rendezvous (of tasks) **9.5**; 9, 9.7.1, 9.7.2, 9.7.3, D
during which an exception is
raised 11.5
priority 9.8
prohibited for an abnormal task
9.10

Replacement of characters in program text **2.10**

Representation (of a type and its objects) **13.1**
recommendation by a pragma
13.1

Representation attribute 13.7.2, **13.7.3**
as a forcing occurrence 13.1
with a prefix that has a null value
4.1

Representation clause 13.1; 13.6, D
[see also: address clause, elaboration
of . . . , enumeration representation
clause, first named subtype, length
clause, record representation clause,
type]
an overload resolution context 8.7
as a basic declarative item 3.9
as a portion of a declarative
region 8.1
cannot include a forcing
occurrence 13.1
for a derived type 3.4
for a private type 7.4.1
implied for a derived type 3.4
in a task specification 9.1
DEC Ada restrictions F

Reserved word 2.9; 2.2, 2.3

RESET (input-output procedure)
in an instance of direct_io 14.2.1;
14.2.5
in an instance of sequential_io
14.2.1; 14.2.3
in text_io 14.2.1; 14.3.1, 14.3.10
in an instance of indexed_io
14.2a.1, 14.2a.5
in an instance of relative_io
14.2.1, 14.2a.3
in direct_mixed_io 14.2.1, 14.2b.6
in indexed_mixed_io 14.2a.1,
14.2b.10
in relative_mixed_io 14.2.1,
14.2b.8
in sequential_mixed_io 14.2.1,
14.2b.4

Resolution of overloading
[see: overloading]

Result subtype (of a function) **6.1**
of a return expression 5.8

Result type profile
[see: parameter and . . .]

**Result type and overload resolution
6.6**

Result of a function
[see: returned value]

Return
[see: carriage return]

Return statement 5.8
[see also: function, statement]
as a simple statement 5.1
causing a loop to be exited 5.5
causing a transfer of control 5.1
completing block statement
execution 9.4
completing subprogram execution
9.4
expression that is an array
aggregate 4.3.2
in a function body 6.5

Returned value
[see: function call]
of a function call 5.8, 6.5; 8.5
of an instance of a generic formal function 12.1.3
of a main program 10.1
of an operation 3.3.3
of a predefined operator of an integer type 3.5.4
of a predefined operator of a real type 3.5.6, 4.5.7

Right label bracket compound delimiter **2.2**

Right parenthesis
character 2.1
delimiter 2.2

Rounding
in a real-to-integer conversion 4.6
of results of real operations 4.5.7; 13.7.3

Run time check 11.7; 11.1

Safe interval 4.5.7

Safe number (of a real type) **3.5.6**; 4.5.7
[see also: model number, real type representation attribute, real type]
limit to the result of a real operation 4.5.7
of a fixed point type 3.5.9; 3.5.10
of a floating point type 3.5.7; 3.5.8
result of universal expression too large 4.10

SAFE_EMAX (predefined attribute) **3.5.8**; A
DEC Ada floating point values for F

SAFE_LARGE (predefined attribute) **3.5.8**, **3.5.10**; A
DEC Ada floating point values for F

SAFE_SMALL (predefined attribute) **3.5.8**, **3.5.10**; A
DEC Ada floating point values for F

Satisfy (a constraint) **3.3**; D
[see also: constraint, subtype]
a discriminant constraint 3.7.2
an index constraint 3.6.1
a range constraint 3.5

Scalar type 3.3, **3.5**; D
[see also: class of type, discrete type, enumeration type, fixed point type, floating point type, integer type, numeric type, real type, static expression]
as a generic parameter 12.1.2, 12.3.3
formal parameter 6.2
of a range in a membership test 4.5.2
operation 3.5.5; 4.5.2

Scheduling 9.8; 13.5.1

Scheme
[see: iteration scheme]

Scope 8.2; 8.3, D
[see also: basic operation, character literal, declaration, declarative region, generic instance, identifier, immediate scope, implicit declaration, operator symbol, overloading, visibility]
of a use clause 8.4

Secondary unit 10.1
[see also: compilation unit, library unit]
compiled after the corresponding library unit or parent unit 10.3
subject to a pragma elaborate 10.5

not allowed in a numeric literal
2.4.1

Space character literal 2.5; 2.2

Special character 2.1
[see also: basic graphic character, other special character]
in a delimiter 2.2

Specification
[see: declaration, discriminant specification, enumeration literal specification, generic specification, loop parameter specification, package specification, parameter specification, subprogram specification, task specification]

STANDARD (predefined package) **8.6**; C
[see also: library unit]
as a declarative region 8.1
enclosing the library units of a program 10.1.1; 10.1, 10.2
including implicit declarations of fixed point cross-multiplication and cross-division 4.5.5
implicit recompilation of 3.5.7a, 3.5.7b

STANDARD_INPUT (text_io function) **14.3.2**; 14.3.10

STANDARD_OUTPUT (text_io function) **14.3.2**; 14.3.10

Star
[see: double star]
character 2.1
delimiter 2.2

Statement 5.1; 5, D
[see also: abort statement, accept statement, address attribute, assignment statement, block statement, case statement, code statement, compound statement, delay statement, entry call statement, exit statement, goto statement, if statement, label, loop statement,

null statement, procedure call statement, raise statement, return statement, select statement, sequence of statements, target statement]
allowed in an exception handler 11.2
as an overload resolution context 8.7
optimized 10.6
raising an exception 11.4.1; 11.4
that cannot be reached 10.6

Statement alternative
[see: case statement alternative]

Static constraint 4.9
on a subcomponent subject to a component clause 13.4
on a type 3.5.4, 3.5.7, 3.5.9, 13.2

Static discrete range 4.9
as a choice of an aggregate 4.3.2
as a choice of a case statement 5.4
as a choice of a variant part 3.7.3

Static expression 4.9; 8.7
as a bound in an integer type definition 3.5.4
as a choice in a case statement 5.4
as a choice of a variant part 3.7.3
for a choice in a record aggregate 4.3.2
for a discriminant in a record aggregate 4.3.1
in an attribute designator 4.1.4
in an enumeration representation clause 13.3
in a fixed accuracy definition 3.5.9
in a floating accuracy definition 3.5.7
in a generic unit 12.1
in a length clause 13.2
in a number declaration 3.2, 3.2.2
in a record representation clause 13.4
in priority pragma 9.8

as a primary 4.4
must not be the argument of a
conversion 4.6
replaced by a catenation of basic
characters 2.10

Stub
[see: body stub]

Subaggregate 4.3.2

Subcomponent 3.3; D
[see also: component, composite type,
default expression, discriminant,
object]
depending on a discriminant
3.7.1; 5.2, 8.5
of a component for which a
component clause is given 13.4
renamed 8.5
that is a task object 9.2; 9.3
whose type is a limited type 7.4.4
whose type is a private type 7.4.1

Subprogram 6; D
[see also: actual parameter, completed
subprogram, derived subprogram,
entry, formal parameter, function,
library unit, overloading of . . . ,
parameter and result type profile,
parameter, predefined subprogram,
procedure, program unit]
as a generic instance 12.3; 12
as a main program 10.1
as an operation 3.3.3; 7.4.2
including a raise statement 11.3
of a derived type 3.4
overloaded 6.6
renamed 8.5
subject to an address clause 13.5
subject to an inline pragma 6.3.2
subject to an interface pragma
13.9
subject to a representation clause
13.1
subject to a suppress pragma 11.7
with a separately compiled body
10.2

exporting Ada to non-Ada
programs 13.9a.1.4
importing non-Ada from Ada
programs 13.9, 13.9a.1.1,
13.9a.1.2, 13.9a.1.3

Subprogram body 6.3; 6, D
[see also: body stub]
as a generic body 12.2
as a library unit 10.1
as a proper body 3.9
as a secondary unit 10.1
as a secondary unit compiled
after the corresponding library
unit 10.3
having dependent tasks 9.4
in a package body 7.1
including an exception handler
11.2; 11
including an exit statement 5.7
including a goto statement 5.9
including an implicit declaration
5.1
including a return statement 5.8
including code statements must
be a procedure body 13.8
inlined in place of each call 6.3.2
must be in the same declarative
region as the declaration 3.9, 7.1
not allowed for a subprogram
subject to an interface pragma
13.9
not yet elaborated at a call 3.9
raising an exception 11.4.1, 11.4.2
recompiled 10.3

Subprogram call 6.4; 6, 6.3, 12.3
[see also: actual parameter, entry call
statement, entry call, function call,
procedure call statement, procedure
call]
before elaboration of the body 3.9
statement replaced by an inlining
of the body 6.3.2
statement with a default actual
parameter 6.4.2
to a derived subprogram 3.4
to a generic instance 12

Subtype indication 3.3.2
[see also: elaboration of . . .]
as a component subtype
indication 3.7
as a discrete range 3.6
for a subtype of a generic formal
type 12.1.2
in an access type definition 3.8
in an allocator 4.8
in an array type definition 3.6
in a component declaration 3.7
in a constrained array definition
3.6
in a derived type definition 3.4
in a generic formal part 12.1
in an object declaration 3.2, 3.2.1
in an unconstrained array
definition 3.6
including a fixed point constraint
3.5.9
including a floating point
constraint 3.5.7
with a range constraint 3.5

Subunit 10.2; D
[see also: library unit]
as a compilation unit 10.4
as a library unit 10.4
as a secondary unit 10.1
compiled after the corresponding
parent unit 10.3
not allowed for a subprogram
subject to an interface pragma
13.9
of a compilation unit subject to a
context clause 10.1.1
raising an exception 11.4.1, 11.4.2
recompiled (does not affect other
compilation units) 10.3

SUCC (predefined attribute) **3.5.5**; 13.3, A

Successor
[see: succ attribute]

SUPPRESS (predefined pragma) **11.7**;
11.1, B

SUPPRESS_ALL (DEC Ada predefined
pragma) **11.7**; B

Symbol
[see: graphical symbol, operator
symbol]

Synchronization of tasks
[see: task synchronization]

Syntactic category 1.5

Syntax notation 1.5

Syntax rule 1.5; E

SYSTEM (predefined library package)
13.7; C, F

System dependent F
attribute 13.4
constant 13.7
named number 13.7, 13.7.1
record component 13.4
type 13.7

System services (OpenVMS)
asynchronous 9.12a
calling from DEC Ada 9.12a
clock interval provided by 13.7.1
using null_parameter (DEC Ada
predefined attribute) in calls to
13.9a.1.3
reading volatile variables from
9.11

SYSTEM.ADD_ATOMIC (DEC Ada
predefined procedure) **13.7b.6**

SYSTEM.ADD_INTERLOCKED (DEC
Ada predefined procedure) **13.7b.5**

SYSTEM.ADDRESS (predefined type)
13.7; 13.5
[see also: address attribute, address
clause]
properties of 13.7a.1

SYSTEM.ADDRESS_SIZE (DEC Ada
predefined constant) **13.7a.1**

SYSTEM_NAME (predefined pragma) **13.7**; B
> [see also: system.system_name predefined constant]

Tabulation
> [see: horizontal tabulation, vertical tabulation]

Target statement (of a goto statement) **5.9**

Target type of a conversion **4.6**

Task 9; D
> [see also: abnormal task, abort statement, accept statement, communication between . . . , completed task, delay statement, dependent task, entry (of a task), entry call statement, rendezvous, select statement, selective wait, shared variable, single task, terminated task]
>> calling the main program 10.1
>> raising an exception 11.5
>> scheduling 9.8, 9.8a
>> suspension awaiting a rendezvous 9.5
>> suspension by a delay statement 9.6
>> suspension by a selective wait 9.7.1
>> suspension of an abnormal task 9.10
>
> [see also: main_storage pragma, task_storage pragma, time_slice pragma]

Task activation 9.3
> [see also: length clause, storage units allocated, storage_size attribute]
>> before elaboration of the body 3.9
>> causing synchronization 9.10, 9.11

not started for an abnormal task 9.10
of a task with no task body 11.1
specification of storage for 13.2a

Task body 9.1; 9, D
> [see also: body stub, elaboration of . . .]
>> as a proper body 3.9
>> in a package body 7.1
>> including an exception handler 11.2; 11
>> including an exit statement 5.7
>> including a goto statement 5.9
>> including an implicit declaration 5.1
>> must be in the same declarative region as the declaration 3.9, 7.1
>> not yet elaborated at an activation 3.9
>> raising an exception 11.4.1, 11.4.2
>> specifying the execution of a task 9.2, 9.3

Task communication
> [see: rendezvous]

Task completion
> [see: completed task]

Task declaration 9.1
>> and body as a declarative region 8.1
>> as a basic declaration 3.1
>> as a later declarative item 3.9
>> elaboration raising an exception 11.4.2
>> in a package specification 7.1

Task dependence
> [see: dependent task]

Task designated
>> by a formal parameter 6.2
>> by a value of a task type 9.1; 9.2, 9.4, 9.5

Task execution 9.3

Terminator
[see: file terminator, line terminator, page terminator]

Text input-output 14.3; 14.2.1

Text of a program **2.2**, **10.1**

TEXT_IO (predefined input-output package) **14.3**; 14, 14.1, 14.3.9, 14.3.10, C
exceptions 14.4; 14.5
specification 14.3.10

TICK
[see: system.tick]

TIME (predefined type) **9.6**
[see also: clock, date, day, make_time, month, system.tick, year]

TIME_ERROR (predefined exception) **9.6**

TIME_OF (predefined function) **9.6**

TIME_SLICE (DEC Ada predefined pragma) **9.8a**; B

Timed entry call 9.7.3; 9.7
and renamed entries 8.5
subject to an address clause 13.5.1

Times operator
[see: multiplying operator]

TITLE (DEC Ada predefined pragma) **B**

TO_ADDRESS (DEC Ada predefined function)
[see: system.to_address]

TO_BIT_ARRAY_8 (DEC Ada predefined function)
[see: system.to_bit_array_8]

TO_BIT_ARRAY_16 (DEC Ada predefined function)
[see: system.to_bit_array_16]

TO_BIT_ARRAY_32 (DEC Ada predefined function)
[see: system.to_bit_array_32]

TO_BIT_ARRAY_64 (DEC Ada predefined function)
[see: system.to_bit_array_64]

TO_INTEGER (DEC Ada predefined function)
[see: system.to_integer]

TO_UNSIGNED_BYTE (DEC Ada predefined function)
[see: system.to_unsigned_byte_]

TO_UNSIGNED_LONGWORD (DEC Ada predefined function)
[see: system.to_unsigned_longword]

TO_UNSIGNED_QUADWORD (DEC Ada predefined function)
[see: system.to_unsigned_quadword]

TO_UNSIGNED_WORD (DEC Ada predefined function)
[see: system.to_unsigned_word]

Transfer of control 5.1
[see also: exception, exit statement, goto statement, return statement, terminate alternative]

TRUE boolean enumeration literal **3.5.3**; C

Type 3.3; D
[see also: access type, appropriate for a type, array type, attribute of . . . , base attribute, base type, boolean type, character type, class of type, composite type, constrained type, derived type, discrete type, discriminant of . . . , enumeration type, fixed point type, floating point type, forcing occurrence, generic actual type, generic formal type, integer type, limited private type, limited type, numeric type, operation of . . . , parent type, predefined type, private type, real type, record type, representation clause, scalar type, size attribute, storage allocated,

in a generic parameter
declaration 12.3.1
in an index subtype definition 3.6
in a parameter specification 6.1;
6.2
in a qualified expression 4.7
in a relation 4.4
in a renaming declaration 8.5
in a subprogram specification 6.1
of a formal parameter of a generic
formal subprogram 12.1.3
of a generic formal array type
12.1.2
of a static scalar subtype 4.9
of the result of a generic formal
function 12.1.3

Type with discriminants 3.3; 3.3.1,
3.3.2, 3.7, 3.7.1, 7.4, 7.4.1
[see also: private type, record type]
as an actual to a formal private
type 12.3.2
as the component type of an
array that is the operand of a
conversion 4.6

TYPE_CLASS (DEC Ada predefined
attribute) **13.7a.2**; A

TYPE_CLASS (DEC Ada predefined type)
[see: system.type_class]

Unary adding operator 4.4, **4.5**, **C**; 4.5.4
[see also: arithmetic operator,
overloading of an operator, predefined
operator]
as an operation of a discrete type
3.5.5
in a simple expression 4.4
overloaded 6.7

Unary operator 4.5; 3.5.5, 3.5.8, 3.5.10,
3.6.2, 4.5.4, 4.5.6, C
[see also: highest precedence operator,
unary adding operator]

UNCHECKED_CONVERSION
(predefined generic library function)
13.10.2; 13.10, C

UNCHECKED_DEALLOCATION
(predefined generic library procedure)
13.10.1; 4.8, 13.10, C

Unconditional termination of a task
[see: abnormal task, abort statement]

Unconstrained array definition 3.6

Unconstrained array type 3.6; 3.2.1
as an actual to a formal private
type 12.3.2
formal parameter 6.2
subject to a length clause 13.2

Unconstrained subtype 3.3, **3.3.2**
[see also: constrained subtype,
constraint, subtype, type]
indication in a generic unit 12.3.2

Unconstrained type 3.3; 3.2.1, 3.6, 3.6.1,
3.7, 3.7.2
formal parameter 6.2
with discriminants 6.4.1, 12.3.2

Unconstrained variable 3.3, 3.6, 3.7;
12.3.1
as a subcomponent [see:
subcomponent]

Undefined value
of a scalar parameter 6.2
of a scalar variable 3.2.1

Underflow (floating point) handling in an
exception 11.1

Underline character 2.1
in a based literal 2.4.2
in a decimal literal 2.4.1
in an identifier 2.3

Unhandled exception 11.4.1

Unit
[see: compilation unit, generic unit, library unit, program unit, storage unit, task unit]

Universal expression 4.10
assigned 5.2
in an attribute designator 4.1.4
of a real type implicitly converted 4.5.7
that is static 4.10

Universal type 4.10
[see also: conversion, implicit conversion]
expression [see: expression, numeric literal]
of a named number 3.2.2; 3.2
result of an attribute [see: attribute]

UNIVERSAL_FIXED (predefined type) **3.5.9**
result of fixed point multiplying operators 4.5.5

UNIVERSAL_INTEGER (predefined type) **3.5.4**, **4.10**; C
[see also: integer literal]
argument of a conversion 3.3.3, 4.6
attribute 3.5.5, 13.7.1, 13.7.2, 13.7.3; 9.9
bounds of a discrete range 3.6.1
bounds of a loop parameter 5.5
codes representing enumeration type values 13.3
converted to an integer type 3.5.5
of integer literals 2.4, 4.2
result of an operation 4.10; 4.5

UNIVERSAL_REAL (predefined type) **3.5.6**, **4.10**
[see also: real literal]
argument of a conversion 3.3.3, 4.6
attribute 13.7.1

converted to a fixed point type 3.5.10
converted to a floating point type 3.5.8
of real literals 2.4, 4.2
result of an operation 4.10; 4.5

UNLOCK (DEC Ada input-output procedure)
in an instance of indexed_io 14.2a.4, 14.2a.5
in an instance of relative_io 14.2a.2, 14.2a.3
in indexed_mixed_io 14.2b.9, 14.2b.10
in relative_mixed_io 14.2b.7, 14.2b.8

UNSIGNED_BYTE (DEC Ada predefined type)
[see: system.unsigned_byte]

UNSIGNED_BYTE_ARRAY (DEC Ada predefined type)
[see: system.unsigned_byte_array]

UNSIGNED_LONGWORD (DEC Ada predefined type)
[see: system.unsigned_longword]

UNSIGNED_LONGWORD_ARRAY (DEC Ada predefined type)
[see: system.unsigned_longword_array]

UNSIGNED_QUADWORD (DEC Ada predefined type)
[see: system.unsigned_quadword]

UNSIGNED_QUADWORD_ARRAY (DEC Ada predefined type)
[see: system.unsigned_quadword_array]

UNSIGNED_WORD (DEC Ada predefined type)
[see: system.unsigned_word]

Variable declaration 3.2.1

Variant 3.7.3; 4.1.3
[see also: component clause, record type]
in a variant part 3.7.3

Variant part 3.7.3; D
[see also: dependence on a discriminant]
in a component list 3.7
in a record aggregate 4.3.1

Vertical bar character 2.1
replacement by exclamation character 2.10

Vertical bar delimiter 2.2

Vertical tabulation format effector **2.1**

Violation of a constraint
[see: constraint_error exception]

Visibility 8.3; 8.2, D
[see also: direct visibility, hiding, identifier, name, operation, overloading]
and renaming 8.5
determining multiple meanings of an identifier 8.4, 8.7; 8.5
determining order of compilation 10.3
due to a use clause 8.4
of a basic operation 8.3
of a character literal 8.3
of a default for a generic formal subprogram 12.3.6
of a generic formal parameter 12.3
of a library unit due to a with clause 8.6, 10.1.1
of a name of an exception 11.2
of an operation declared in a package 7.4.2
of an operator symbol 8.3
of a renaming declaration 8.5
of a subprogram declared in a package 6.3

of declarations in a package body 7.3
of declarations in a package specification 7.2
of declarations in the package system 13.7
within a subunit 10.2

Visibility by selection 8.3
[see also: basic operation, character literal, operation, operator symbol, selected component]

Visible part (of a package) **7.2**; 3.2.1, 7.4, 7.4.1, 7.4.3, D
[see also: deferred constant declaration, private type declaration]
expanded name denoting a declaration in a visible part 8.2
scope of a declaration in a visible part 4.1.3
use clause naming the package 8.4
visibility of a declaration in a visible part 8.3

VOLATILE (DEC Ada predefined pragma) **9.11**; B

Wait
[see: selective wait, task suspension]

While loop
[see: loop statement]

WIDTH (predefined attribute) **3.5.5**; A

With clause 10.1.1; D
[see also: context clause]
determining order of compilation 10.3
determining the implicit order of library units 8.6

Postscript : Submission of Comments

---------- **Note** ----------

This postscript is not part of the standard definition of the Ada programming language.

For submission of comments on the DEC Ada information (all material printed in color), use the Reader's Comments form at the back of the manual. For reporting errors or problems encountered when using the DEC Ada software, submit a Software Problem Report (SPR).

For submission of comments on the standard Ada reference manual (all material printed in black), use the Arpanet address

Ada-Comment at ECLB

If you do not have Arpanet access, please send the comments on the standard Ada reference manual by mail to

Ada Joint Program Office (AJPO)
Office of the Under Secretary of Defense Research and Engineering
Washington, DC 20301
United States of America

If you mail your comments, it will assist the AJPO if you can send them on 8-inch single-sided single-density IBM format diskettes—with an additional paper copy, in case of problems with reading the diskettes.

All comments are sorted and processed mechanically, in order to simplify their analysis and to facilitate giving them proper consideration. To aid this process, please precede each comment with the three line header

!section . . .
!version 1983
!topic . . .

The section line should include the section number, the paragraph number enclosed in parentheses, your name or affiliation (or both), and the date in ISO standard form (year-month-day). The paragraph number is the one given in the margin of the paper form of this document (it is not contained in the ECLB files); paragraph numbers are optional, but very helpful. For example, here is the section line of comment #1194 on a previous version:

!section 03.02.01(12)D.Taffs 82–04–26

The version line for comments on the current standard should contain only "!version 1983". The purpose of this line is to distinguish comments that refer to different versions.

The topic line should contain a one line summary of the comment. This line is essential, and you are kindly asked to avoid topics such as "Typo" or "Editorial comment", which will not convey any information when printed in a table of contents. As an example of an informative topic line consider:

> !topic Subcomponents of constants are constants

Nothing prevents the topic line from including all the information of a comment, as in the following topic line:

> !topic Insert: " . . . are {implicitly} defined by a subtype declaration"

As a final example here is a complete comment received on a prior version of this manual:

> !section 03.02.01(12)D.Taffs 82–04–26
> !version 10
> !topic Subcomponents of constants are constants
>
> Change "component" to "subcomponent" in the last sentence.
>
> Otherwise the statement is inconsistent with the defined use of subcomponent
> in 3.3, which says that subcomponents are excluded when the term component is used instead of subcomponent.