ISO/IEC JTC 1/SC 22/WG 9 N 467

To: Secretariat, ISO/IEC JTC 1/SC 22
From: Convener, ISO/IEC JTC 1/SC 22/WG 9
Subject: Submission of ISO/IEC 8652:1995/FPDAM 1 to SC22 for FPDAM ballot
Date: 27 April 2006

I am submitting the attached document for balloting as authorized by Resolution 05-29 of the 2005 plenary meeting of SC 22:

Resolution 05-29: Authorization to Initiate an FPDAM Ballot for Ada

JTC 1/SC 22 instructs its Secretariat to initiate an FPDAM ballot for ISO/IEC 8652: 1995, Ada, upon receipt of the text from the WG 9 (Ada) Convener.

I request that balloting commence as soon as is convenient so that any necessary comment disposition could be performed at the SC22 plenary meeting.

Thank you.

ISO/IEC JTC 1/SC 22 N

Date: 2006-04-26

ISO/IEC 8652:1995/FPDAM 1

ISO/IEC JTC 1/SC 22/WG 9

Information Technology —Programming languages — Ada

AMENDMENT 1

Technologies de l'information —Langages de programmation — Ada

AMENDEMENT 1

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Copyright notice

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Amendment 1 to International Standard ISO/IEC 8652:1995 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*, Subcommittee SC 22, *Programming Languages, their environments and system software interfaces*.

Amendment 1 cancels and replaces those portions of International Standard ISO/IEC 8652:1995 as corrected by Technical Corrigendum 1 ISO/IEC 8652:1995/COR.1:2001 as specified by the body of this amendment. Those portions of the International Standard as corrected by Technical Corrigendum 1 not modified by this amendment remain in force.

The main emphasis of Amendment is to improve the object-oriented programming and the real-time features of International Standard ISO/IEC 8652:1995 while also strengthening reliability.

As in International Standard ISO/IEC 8652:1995, Annexes A to J form an integral part of the standard as amended. Annexes K to Q are for information only.

Introduction

International Standard ISO/IEC 8652:1995 defines the Ada programming language.

This amendment modifies Ada by making changes and additions that improve:

- The safety of applications written in Ada;
- The portability of applications written in Ada;
- Interoperability with other languages and systems; and
- Accessibility and ease of transition from idioms in other programming and modeling languages.

This amendment incorporates the following major additions to the International Standard:

- Support for the entire ISO/IEC 10646:2003 character repertoire, both in program text and executing programs (see clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4);
- Interfaces, to provide a limited form of multiple inheritance of operations (see clause 3.9.4);
- Improvements for access types, such as null excluding subtypes (see clause 3.10), additional uses for anonymous access types (see clauses 3.6 and 8.5.1), and anonymous access-to-subprogram subtypes to support 'downward closures' (see clauses 3.10 and 3.10.2);
- Additional context clause capabilities: limited views to allow mutually dependent types (see clauses 3.10.1 and 10.1.2) and private with clauses that apply only in the private part of a package (see clause 10.1.2);
- Aggregates, constructor functions, and constants for limited types (see clauses 4.3.1, 6.5, and 7.5);
- Control of overriding to eliminate errors (see clause 8.3);
- Additional standard packages, including time management (see 9.6), file directory and name management (see clause A.16), containers (see clause A.18), execution-time clocks (see clause D.14), timing events (see clause D.15), and vector and matrix operations (see clause G.3);
- A mechanism for writing C unions to make interfaces with C systems easier (see clause B.3.3);
- New task dispatching policies, including non-preemptive (see clause D.2.4) and earliest deadline first (see clause D.2.6); and
- The Ravenscar profile to provide a simplified tasking system for high-integrity systems (see clause D.13).

This Amendment is organized by sections corresponding to those in the International Standard. These sections include wording changes and additions to the International Standard. Clause and subclause headings are given for each clause that contains a wording change. Clauses and subclauses that do not contain any change or addition are omitted.

For each change, an *anchor* paragraph from the International Standard (as corrected by Technical Corrigendum 1) is given. New or revised text and instructions are given with each change. The anchor paragraph can be replaced or deleted, or text can be inserted before or after it. When a heading immediately precedes the anchor paragraph, any text inserted before the paragraph is intended to appear under the heading.

Typographical conventions:

Instructions about the text changes are in this font. The actual text changes are in the same fonts as the International Standard - this font for text, this font for syntax, `and this font for Ada source code.`

Introduction

Of International Standard ISO/IEC 8652:1995. Modifications of this section of that standard are found here.

Replace paragraph 3:

- Rationale for the Ada Programming Language -- 1995 edition, which gives an introduction to the new features of Ada, and explains the rationale behind them. Programmers should read this first.

by:

- Ada 95 Rationale. This gives an introduction to the new features of Ada incorporated in the 1995 edition of this Standard, and explains the rationale behind them. Programmers unfamiliar with Ada 95 should read this first.
- Ada 2005 Rationale. This gives an introduction to the changes and new features in Ada 2005 (compared with the 1995 edition), and explains the rationale behind them. Programmers should read this rationale before reading this Standard in depth.

Replace paragraph 5:

- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the RM95, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

by:

- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the consolidated Ada Reference Manual, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

Replace paragraph 6:

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. This revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency.

by:

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. The 1995 revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency. This amended version provides further flexibility and adds more standardized packages within the framework provided by the 1995 revision.

Replace paragraph 32:

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, Character, and Wide_Character are predefined.

by:

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, Character, Wide_Character, and Wide_Wide_Character are predefined.

Replace paragraph 34:

Composite types allow definitions of structured objects with related components. The composite types in the language include 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. Task and protected types are also forms of composite types. The array types String and Wide_String are predefined.

by:

Composite types allow definitions of structured objects with related components. The composite types in the language include 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. Task and protected types are also forms of composite types. The array types String, Wide_String, and Wide_Wide_String are predefined.

Insert after paragraph 38:

From any type a new type may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand.

the new paragraph:

Interface types provide abstract models from which other interfaces and types may be composed and derived. This provides a reliable form of multiple inheritance. Interface types may also be implemented by task types and protected types thereby enabling concurrent programming and inheritance to be merged.

Replace paragraph 41:

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.

by:

Aspect 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.

Replace paragraph 42:

The predefined environment of the language provides for input-output and other capabilities (such as string manipulation and random number generation) by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided. Other standard library packages are defined in annexes of the standard to support systems with specialized requirements.

by:

The predefined environment of the language provides for input-output and other capabilities by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

The predefined standard library packages provide facilities such as string manipulation, containers of various kinds (vectors, lists, maps, etc.), mathematical functions, random number generation, and access to the execution environment.

The specialized annexes define further predefined library packages and facilities with emphasis on areas such as real-time scheduling, interrupt handling, distributed systems, numerical computation, and high-integrity systems.

Replace paragraph 44:

This International Standard replaces the first edition of 1987. In this edition, the following major language changes have been incorporated:

by:

This amended International Standard updates the edition of 1995 which replaced the first edition of 1987. In the 1995 edition, the following major language changes were incorporated:

Replace paragraph 45:

- Support for standard 8-bit and 16-bit character sets. See Section 2, 3.5.2, 3.6.3, A.1, A.3, and A.4.

by:

- Support for standard 8-bit and 16-bit characters was added. See clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4.

Replace paragraph 46:

- Object-oriented programming with run-time polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. See also the new forms of generic formal parameters that are allowed by 12.5.1, "Formal Private and Derived Types" and 12.7, "Formal Packages".

by:

- The type model was extended to include facilities for object-oriented programming with dynamic polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. Additional forms of generic formal parameters were allowed as described in clauses 12.5.1 and 12.7.

Replace paragraph 47:

- Access types have been extended to allow an access value to designate a subprogram or an object declared by an object declaration (as opposed to just a heap-allocated object). See 3.10.

by:

- Access types were extended to allow an access value to designate a subprogram or an object declared by an object declaration as opposed to just an object allocated on a heap. See clause 3.10.

Replace paragraph 48:

- Efficient data-oriented synchronization is provided via protected types. See Section 9.

by:

- Efficient data-oriented synchronization was provided by the introduction of protected types. See clause 9.4.

Replace paragraph 49:

- The library units of a library may be organized into a hierarchy of parent and child units. See Section 10.

by:

- The library structure was extended to allow library units to be organized into a hierarchy of parent and child units. See clause 10.1.

Replace paragraph 50:

- Additional support has been added for interfacing to other languages. See Annex B.

by:

- Additional support was added for interfacing to other languages. See Annex B.

Replace paragraph 51:

- The Specialized Needs Annexes have been added to provide specific support for certain application areas:

by:

- The Specialized Needs Annexes were added to provide specific support for certain application areas:

Replace paragraph 57:

- Annex H, "Safety and Security"

by:

- Annex H, "High Integrity Systems"

Amendment 1 modifies the 1995 International Standard by making changes and additions that improve the capability of the language and the reliability of programs written in the language. In particular the changes were designed to improve the portability of programs, interfacing to other languages, and both the object-oriented and real-time capabilities.

The following significant changes with respect to the 1995 edition are incorporated:

- Support for program text is extended to cover the entire ISO/IEC 10646:2003 repertoire. Execution support now includes the 32-bit character set. See clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4.
- The object-oriented model has been improved by the addition of an interface facility which provides multiple inheritance and additional flexibility for type extensions. See clauses 3.4, 3.9, and 7.3. An alternative notation for calling operations more akin to that used in other languages has also been added. See clause 4.1.3.
- Access types have been further extended to unify properties such as the ability to access constants and to exclude null values. See clause 3.10. Anonymous access types are now permitted more freely and anonymous access-to-subprogram types are introduced. See clauses 3.3, 3.6, 3.10, and 8.5.1.
- The control of structure and visibility has been enhanced to permit mutually dependent references between units and finer control over access from the private part of a package. See clauses 3.10.1 and 10.1.2. In addition, limited types have been made more useful by the provision of aggregates, constants, and constructor functions. See clauses 4.3, 6.5, and 7.5.
- The predefined environment has been extended to include additional time and calendar operations, improved string handling, a comprehensive container library, file and directory management, and access to environment variables. See clauses 9.6.1, A.4, A.16, A.17, and A.18.
- Two of the Specialized Needs Annexes have been considerably enhanced:
  - The Real-Time Systems Annex now includes the Ravenscar profile for high-integrity systems, further dispatching policies such as Round Robin and Earliest Deadline First, support for timing events, and support for control of CPU time utilization. See clauses D.2, D.13, D.14, and D.15.
  - The Numerics Annex now includes support for real and complex vectors and matrices as previously defined in ISO/IEC 13813:1997 plus further basic operations for linear algebra. See clause G.3.
- The overall reliability of the language has been enhanced by a number of improvements. These include new syntax which detects accidental overloading, as well as pragmas for making assertions and giving better control over the suppression of checks. See clauses 6.1, 11.4.2, and 11.5.

Section 1: General

1.1.2 Structure

Replace paragraph 13:

- Annex H, ``Safety and Security"

by:

- Annex H, ``High Integrity Systems"

1.1.4 Method of Description and Syntax Notation

Replace paragraph 9:

return_statement ::= **return** [expression];

return_statement ::= **return**; | **return** expression;

by:

simple_return_statement ::= **return** [expression];

simple_return_statement ::= **return**; | **return** expression;

Insert after paragraph 14:

- 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 *subtype*_name and *task*_name are both equivalent to name alone.

the new paragraph:

The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this International Standard (see 2.1) belong to the same range. For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".

Insert before paragraph 15:

A *syntactic category* is a nonterminal in the grammar defined in BNF under "Syntax." Names of syntactic categories are set in a different font, like_this.

the new paragraph:

When this International Standard mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using locale-independent full case folding, as defined by documents referenced in the note in section 1 of ISO/IEC 10646:2003.

1.2 Normative References

Replace paragraph 3:

ISO/IEC 1539:1991, *Information technology — Programming languages — FORTRAN*.

by:

ISO/IEC 1539-1:2004, *Information technology — Programming languages — Fortran — Part 1: Base language*.

Replace paragraph 4:

ISO 1989:1985, *Programming languages — COBOL*.

by:

ISO/IEC 1989:2002, *Information technology — Programming languages — COBOL*.

Insert after paragraph 5:

ISO/IEC 6429:1992, *Information technology — Control functions for coded graphic character sets.*

the new paragraph:

ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

Replace paragraph 7:

ISO/IEC 9899:1990, *Programming languages — C*.

by:

ISO/IEC 9899:1999, *Programming languages — C*, supplemented by Technical Corrigendum 1:2001 and Technical Corrigendum 2:2004.

Replace paragraph 8:

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*, supplemented by Technical Corrigendum 1:1996.

by:

ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

ISO/IEC 14882:2003, *Programming languages — C++*.

ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*.

1.3 Definitions

Replace paragraph 1:

Terms are defined throughout this International Standard, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given in Annex N, "Glossary".

by:

Terms are defined throughout this International Standard, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Mathematical terms not defined in this International Standard are to be interpreted according to the *CRC Concise Encyclopedia of Mathematics, Second Edition*. Other terms not defined in this International Standard are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given in Annex N, "Glossary".

Section 2: Lexical Elements

2.1 Character Set

Replace paragraph 1:

The only characters allowed outside of comments are the graphic_characters and format_effectors.

by:

The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2003 Universal Multiple-Octet Coded Character Set. This coding space is organized in *planes*, each plane comprising 65536 characters.

Delete paragraph 2:

character ::= graphic_character | format_effector | other_control_function

Replace paragraph 3:

graphic_character ::= identifier_letter | digit | space_character | special_character

by:

A character is defined by this International Standard for each cell in the coding space described by ISO/IEC 10646:2003, regardless of whether or not ISO/IEC 10646:2003 allocates a character to that cell.

Replace paragraph 4:

The character repertoire for the text of an Ada program consists of the collection of characters called the Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set, plus a set of format_effectors and, in comments only, a set of other_control_functions; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1).

by:

The coded representation for characters is implementation defined (it need not be a representation defined within ISO/IEC 10646:2003). A character whose relative code position in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program.

The semantics of an Ada program whose text is not in Normalization Form KC (as defined by section 24 of ISO/IEC 10646:2003) is implementation defined.

Replace paragraph 5:

The description of the language definition in this International Standard uses the graphic symbols defined for Row 00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this International Standard for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

by:

The description of the language definition in this International Standard uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by the note in section 1 of ISO/IEC 10646:2003. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

Replace paragraph 6:

The categories of characters are defined as follows:

by:

Characters are categorized as follows:

Delete paragraph 7:

identifier_letter
upper_case_identifier_letter | lower_case_identifier_letter

Replace paragraph 8:

upper_case_identifier_letter
Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Capital Letter".

by:

letter_uppercase
Any character whose General Category is defined to be "Letter, Uppercase".

Replace paragraph 9:

lower_case_identifier_letter
Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Small Letter".

by:

letter_lowercase
Any character whose General Category is defined to be "Letter, Lowercase".

letter_titlecase
Any character whose General Category is defined to be "Letter, Titlecase".

letter_modifier
Any character whose General Category is defined to be "Letter, Modifier".

letter_other
Any character whose General Category is defined to be "Letter, Other".

mark_non_spacing
Any character whose General Category is defined to be "Mark, Non-Spacing".

mark_spacing_combining
Any character whose General Category is defined to be "Mark, Spacing Combining".

Replace paragraph 10:

digit
One of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

by:

number_decimal
Any character whose General Category is defined to be "Number, Decimal".

number_letter
Any character whose General Category is defined to be "Number, Letter".

punctuation_connector
Any character whose General Category is defined to be "Punctuation, Connector".

other_format
Any character whose General Category is defined to be "Other, Format".

Replace paragraph 11:

space_character
The character of ISO 10646 BMP named "Space".

by:

separator_space
Any character whose General Category is defined to be "Separator, Space".

Replace paragraph 12:

special_character
: Any character of the ISO 10646 BMP that is not reserved for a control function, and is not the space_character, an identifier_letter, or a digit.

by:

separator_line
: Any character whose General Category is defined to be "Separator, Line".

separator_paragraph
: Any character whose General Category is defined to be "Separator, Paragraph".

Replace paragraph 13:

format_effector
: The control functions of ISO 6429 called character tabulation (HT), line tabulation (VT), carriage return (CR), line feed (LF), and form feed (FF).

by:

format_effector
: The characters whose code positions are 16#09# (CHARACTER TABULATION), 16#0A# (LINE FEED), 16#0B# (LINE TABULATION), 16#0C# (FORM FEED), 16#0D# (CARRIAGE RETURN), 16#85# (NEXT LINE), and the characters in categories separator_line and separator_paragraph.

other_control
: Any character whose General Category is defined to be "Other, Control", and which is not defined to be a format_effector.

other_private_use
: Any character whose General Category is defined to be "Other, Private Use".

other_surrogate
: Any character whose General Category is defined to be "Other, Surrogate".

Replace paragraph 14:

other_control_function
: Any control function, other than a format_effector, that is allowed in a comment; the set of other_control_functions allowed in comments is implementation defined.

by:

graphic_character
: Any character that is not in the categories other_control, other_private_use, other_surrogate, format_effector, and whose relative code position in its plane is neither 16#FFFE# nor 16#FFFF#.

Replace paragraph 15:

The following names are used when referring to certain special_characters:

by:

The following names are used when referring to certain characters (the first name is that given in ISO/IEC 10646:2003):

Delete paragraph 16:

In a nonstandard mode, the implementation may support a different character repertoire; in particular, the set of characters that are considered identifier_letters can be extended or changed to conform to local conventions.

Replace paragraph 17:

1 Every code position of ISO 10646 BMP that is not reserved for a control function is defined to be a graphic_character by this International Standard. This includes all code positions other than 0000 - 001F, 007F - 009F, and FFFE - FFFF.

by:

1 The characters in categories other_control, other_private_use, and other_surrogate are only allowed in comments.

2.2 Lexical Elements, Separators, and Delimiters

Replace paragraph 2:

The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than character tabulation (HT) signifies at least one end of line.

by:

The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than the character whose code position is 16#09# (CHARACTER TABULATION) signifies at least one end of line.

Replace paragraph 3:

In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format effector, or the end of a line, as follows:

by:

In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a separator_space, a format_effector, or the end of a line, as follows:

Replace paragraph 4:

- A space character is a separator except within a comment, a string_literal, or a character_literal.

by:

- A separator_space is a separator except within a comment, a string_literal, or a character_literal.

Replace paragraph 5:

- Character tabulation (HT) is a separator except within a comment.

by:

- The character whose code position is 16#09# (CHARACTER TABULATION) is a separator except within a comment.

Replace paragraph 8:

A *delimiter* is either one of the following special characters

by:

A *delimiter* is either one of the following characters:

2.3 Identifiers

Replace paragraph 2:

```
identifier ::=
  identifier_letter {[underline] letter_or_digit}
```

by:

```
identifier ::=
  identifier_start {identifier_start | identifier_extend}
```

Replace paragraph 3:

```
letter_or_digit ::= identifier_letter | digit
```

by:

```
identifier_start ::=
    letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter

identifier_extend ::=
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
  | other_format
```

Replace paragraph 4:

An identifier shall not be a reserved word.

by:

After eliminating the characters in category other_format, an identifier shall not contain two consecutive characters in category punctuation_connector, or end with a character in that category.

Replace paragraph 5:

All characters of an identifier are significant, including any underline character. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

by:

Two identifiers are considered the same if they consist of the same sequence of characters after applying the following transformations (in this order):

- The characters in category other_format are eliminated.
- The remaining sequence of characters is converted to upper case.

Replace paragraph 6:

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

by:

After applying these transformations, an identifier shall not be identical to a reserved word (in upper case).

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

NOTES

3 Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

Replace paragraph 8:

```
Count      X    Get_Symbol   Ethelyn   Marion

Snobol_4   X1   Page_Count   Store_Next_Item
```

by:

```
Count      X    Get_Symbol   Ethelyn   Marion
Snobol_4   X1   Page_Count   Store_Next_Item
Πλάτων        -- Plato
Чайковский    -- Tchaikovsky
θ  φ          -- Angles
```

2.4.1 Decimal Literals

Insert after paragraph 5:

exponent ::= E [+] numeral | E - numeral

the new paragraph:

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2.5 Character Literals

Replace paragraph 5:

```
'A'     '*'     '''     ' '
```

by:

```
'A'     '*'     '''     ' '
'L'     'Л'     'Λ'     -- Various els.
'∞'     'א'             -- Big numbers - infinity and aleph.
```

2.6 String Literals

Insert after paragraph 7:

NOTES

6 An end of line cannot appear in a string_literal.

the new paragraph:

7 No transformation is performed on the sequence of characters of a string_literal.

Replace paragraph 9:

```
"Message of the day:"

""                      -- a null string literal
" "   "A"   """"        -- three string literals of length 1

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

by:

```
"Message of the day:"

""                      -- a null string literal
" "   "A"   """"        -- three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
"Archimedes said ""Εύρηκα"""
"Volume of cylinder (πr²h) = "
```

2.8 Pragmas

Replace paragraph 29:

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Inline(Set_Mask); -- generate code for Set_Mask inline
pragma Suppress(Range_Check, On => Index); -- turn off range checking on Index
```

by:

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Inline(Set_Mask); -- generate code for Set_Mask inline
```

```
pragma Import(C, Put_Char, External_Name => "putchar"); -- import C putchar function
```

2.9 Reserved Words

In paragraph 2 replace:

The following are the *reserved words* (ignoring upper/lower case distinctions):

by:

The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case, and one or more characters in category other_format may be inserted within or at the end of the reserved word.

In the list in paragraph 2, add:

interface

overriding

synchronized

Section 3: Declarations and Types

3.1 Declarations

Replace paragraph 3:

```
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | package_declaration | renaming_declaration
  | exception_declaration | generic_declaration
  | generic_instantiation
```

by:

```
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | null_procedure_declaration | package_declaration
  | renaming_declaration | exception_declaration
  | generic_declaration | generic_instantiation
```

Replace paragraph 6:

Each of the following is defined to be a declaration: any basic_declaration; an enumeration_literal_specification; a discriminant_specification; a component_declaration; a loop_parameter_specification; a parameter_specification; a subprogram_body; an entry_declaration; an entry_index_specification; a choice_parameter_specification; a generic_formal_parameter_declaration.

by:

Each of the following is defined to be a declaration: any basic_declaration; an enumeration_literal_specification; a discriminant_specification; a component_declaration; a loop_parameter_specification; a parameter_specification; a subprogram_body; an entry_declaration; an entry_index_specification; a choice_parameter_specification; a generic_formal_parameter_declaration. In addition, an extended_return_statement is a declaration of its defining_identifier.

3.2 Types and Subtypes

Replace paragraph 2:

Types are grouped into *classes* of types, reflecting the similarity of their values and primitive operations. There exist several *language-defined classes* of types (see NOTES below). *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

by:

Types are grouped into *categories* of types. There exist several *language-defined categories* of types (see NOTES below), reflecting the similarity of their values and primitive operations. Most categories of types form *classes* of types. *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

Replace paragraph 4:

The composite types are the *record* types, *record extensions*, *array* types, *task* types, and *protected* types. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. A partial view is a composite type.

by:

The composite types are the *record* types, *record extensions*, *array* types, *interface* types, *task* types, and *protected* types.

There can be multiple views of a type with varying sets of operations. An *incomplete* type represents an incomplete view (see 3.10.1) of a type with a very restricted usage, providing support for recursive data structures. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. The full view (see 3.2.1) of a type represents its complete definition. An incomplete or partial view is considered a composite type, even if the full view is not.

Replace paragraph 5:

Certain composite types (and partial views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type.

by:

Certain composite types (and views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type.

Replace paragraph 6:

The term *subcomponent* is used in this International Standard in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents.

by:

The term *subcomponent* is used in this International Standard in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents. The terms component, subcomponent, and part are also applied to a type meaning the component, subcomponent, or part of objects and values of the type.

Replace paragraph 7:

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 of a *null constraint* that specifies no restriction is also included); the rules for which values satisfy a given kind of constraint are given in 3.5 for range_constraints, 3.6.1 for index_constraints, and 3.7.1 for discriminant_constraints.

by:

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 of a *null constraint* that specifies no restriction is also included); the rules for which values satisfy a given kind of constraint are given in 3.5 for range_constraints, 3.6.1 for index_constraints, and 3.7.1 for discriminant_constraints. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see 3.10).

Replace paragraph 8:

A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the type *of* the subtype. Similarly, the associated constraint is called the constraint *of* the subtype. The set of values of a subtype consists of the values of its type that satisfy its constraint. Such values *belong* to the subtype.

by:

A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the *type of the subtype*. Similarly, the associated constraint is called the *constraint of the subtype*. The set of values of a subtype consists of the values of its type that satisfy its constraint and any exclusion of the null value. Such values *belong* to the subtype.

Replace paragraph 10:

2 Any set of types that is closed under derivation (see 3.4) can be called a "class" of types. However, only certain classes are used in the description of the rules of the language —generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 12.5). The following are examples of "interesting" *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric,

access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes.

by:

2 Any set of types can be called a "category" of types, and any set of types that is closed under derivation (see 3.4) can be called a "class" of types. However, only certain categories and classes are used in the description of the rules of the language —generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 12.5). The following are examples of "interesting" *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes. In addition to these classes, the following are examples of "interesting" *language-defined categories*: abstract, incomplete, interface, limited, private, record.

Replace paragraph 11:

These language-defined classes are organized like this:

by:

These language-defined categories are organized like this:

Replace paragraph 12:

all types
  elementary
    scalar
      discrete
        enumeration
          character
          boolean
          other enumeration
        integer
          signed integer
          modular integer
      real
        floating point
        fixed point
          ordinary fixed point
          decimal fixed point
    access
      access-to-object
      access-to-subprogram
    composite
      array
        string
        other array
      untagged record
      tagged
      task

protected

by:

all types
elementary
scalar
discrete
enumeration
character
boolean
other enumeration
integer
signed integer
modular integer
real
floating point
fixed point
ordinary fixed point
decimal fixed point
access
access-to-object
access-to-subprogram
composite
untagged
array
string
other array
record
task
protected
tagged (including interfaces)
nonlimited tagged record
limited tagged
limited tagged record
synchronized tagged
tagged task
tagged protected

Replace paragraph 13:

The classes "numeric" and "nonlimited" represent other classification dimensions and do not fit into the above strictly hierarchical picture.

by:

There are other categories, such as "numeric" and "discriminated", which represent other categorization dimensions, but do not fit into the above strictly hierarchical picture.

3.2.1 Type Declarations

Replace paragraph 4:

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

by:

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

Replace paragraph 7:

A type defined by a type_declaration is a *named* type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object (including a parameter or a discriminant). The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this International Standard 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. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply "the type T."

by:

A type defined by a type_declaration is a *named* type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object. The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this International Standard 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. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply "the type T".

Replace paragraph 8:

A named type that is declared by a full_type_declaration, or an anonymous type that is defined as part of declaring an object of the type, is called a *full type*. The type_definition, task_definition, protected_definition, or access_definition that defines a full type is called a *full type definition*. Types declared by other forms of type_declaration are not separate types; they are partial or incomplete views of some full type.

by:

A named type that is declared by a full_type_declaration, or an anonymous type that is defined by an access_definition or as part of declaring an object of the type, is called a *full type*. The declaration of a full type also declares the *full view* of the type. The type_definition, task_definition, protected_definition, or access_definition that defines a full type is called a *full type definition*. Types declared by other forms of type_declaration are not separate types; they are partial or incomplete views of some full type.

3.2.2 Subtype Declarations

Replace paragraph 3:

```
subtype_indication ::=  subtype_mark [constraint]
```

by:

```
subtype_indication ::=  [null_exclusion] subtype_mark [constraint]
```


Replace paragraph 15:

```
subtype Rainbow   is Color range Red .. Blue;        -- see 3.2.1
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 3.2.1
subtype Square    is Matrix(1 .. 10, 1 .. 10);       -- see 3.6
subtype Male      is Person(Sex => M);               -- see 3.10.1
```

by:

```
subtype Rainbow   is Color range Red .. Blue;        -- see 3.2.1
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 3.2.1
subtype Square    is Matrix(1 .. 10, 1 .. 10);       -- see 3.6
subtype Male      is Person(Sex => M);               -- see 3.10.1
subtype Binop_Ref is not null Binop_Ptr;             -- see 3.10
```

3.2.3 Classification of Operations

Replace paragraph 1:

An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

by:

An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter or access result type (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

Replace paragraph 7:

- Any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see 8.3) other implicitly declared primitive subprograms of the type.

by:

- For a nonformal type, any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see 8.3) other implicitly declared primitive subprograms of the type.

3.3 Objects and Named Numbers

Replace paragraph 10:

- the result of evaluating a function_call (or the equivalent operator invocation — see 6.6);

by:

- the return object created as the result of evaluating a function_call (or the equivalent operator invocation — see 6.6);

3.3.1 Object Declarations

Replace paragraph 2:

object_declaration ::=
  defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]
 | single_task_declaration
 | single_protected_declaration

by:

object_declaration ::=
  defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]
 | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression]
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]
 | single_task_declaration
 | single_protected_declaration

Replace paragraph 5:

An object_declaration without the reserved word **constant** declares a variable object. If it has a subtype_indication or an array_type_definition that defines an indefinite subtype, then there shall be an initialization expression. An initialization expression shall not be given if the object is of a limited type.

by:

An object_declaration without the reserved word **constant** declares a variable object. If it has a subtype_indication or an array_type_definition that defines an indefinite subtype, then there shall be an initialization expression.

Replace paragraph 8:

The subtype_indication or full type definition of an object_declaration defines the nominal subtype of the object. The object_declaration declares an object of the type of the nominal subtype.

by:

The subtype_indication, access_definition, or full type definition of an object_declaration defines the nominal subtype of the object. The object_declaration declares an object of the type of the nominal subtype.

A component of an object is said to *require late initialization* if it has an access discriminant value constrained by a per-object expression, or if it has an initialization expression that includes a name denoting the current instance of the type or denoting an access discriminant.

Replace paragraph 9:

If a composite object declared by an object_declaration has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant or aliased (see 3.10) the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be *constrained by its initial value*. In the case of an aliased object, this initial value may be either explicit or implicit; in the other cases, an explicit initial value is required. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*.

by:

If a composite object declared by an object_declaration has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be *constrained by its initial value*. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*.

Replace paragraph 16:

1. The subtype_indication, array_type_definition, single_task_declaration, or single_protected_declaration is first elaborated. This creates the nominal subtype (and the anonymous type in the latter three cases).

by:

1. The subtype_indication, access_definition, array_type_definition, single_task_declaration, or single_protected_declaration is first elaborated. This creates the nominal subtype (and the anonymous type in the last four cases).

Replace paragraph 18:

3. The object is created, and, if there is not an initialization expression, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype.

by:

3. The object is created, and, if there is not an initialization expression, the object is *initialized by default*. When an object is initialized by default, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called.

Delete paragraph 19:

4. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called.

Replace paragraph 20:

For the third step above, the object creation and any elaborations and evaluations are performed in an arbitrary order, except that if the default_expression for a discriminant is evaluated to obtain its initial value, then this evaluation is performed before that of the default_expression for any component that depends on the discriminant, and also before that of any default_expression that includes the name of the discriminant. The evaluations of the third step and the assignments of the fourth step are performed in an arbitrary order, except that each evaluation is performed before the resulting value is assigned.

by:

For the third step above, evaluations and assignments are performed in an arbitrary order subject to the following restrictions:

- Assignment to any part of the object is preceded by the evaluation of the value that is to be assigned.
- The evaluation of a default_expression that includes the name of a discriminant is preceded by the assignment to that discriminant.
- The evaluation of the default_expression for any component that depends on a discriminant is preceded by the assignment to that discriminant.
- The assignments to any components, including implicit components, not requiring late initialization must precede the initial value evaluations for any components requiring late initialization; if two components both require late initialization, then assignments to parts of the component occurring earlier in the order of the component declarations must precede the initial value evaluations of the component occurring later.

Replace paragraph 27:

```
John, Paul : Person_Name := new Person(Sex => M);  -- see 3.10.1
```

by:

```
John, Paul : not null Person_Name := new Person(Sex => M);  -- see 3.10.1
```

Replace paragraph 29:

```
John : Person_Name := new Person(Sex => M);
Paul : Person_Name := new Person(Sex => M);
```

by:

```
John : not null Person_Name := new Person(Sex => M);
```

```
Paul : not null Person_Name := new Person(Sex => M);
```

Replace paragraph 31:

```
Count, Sum  : Integer;
Size        : Integer range 0 .. 10_000 := 0;
Sorted      : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option      : Bit_Vector(1 .. 10) := (others => True);
Hello       : constant String := "Hi, world.";
```

by:

```
Count, Sum  : Integer;
Size        : Integer range 0 .. 10_000 := 0;
Sorted      : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option      : Bit_Vector(1 .. 10) := (others => True);
Hello       : aliased String := "Hi, world.";
θ, φ        : Float range -π .. +π;
```

Replace paragraph 33:

```
Limit     : constant Integer := 10_000;
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Real := Dispersion(1.15);
```

by:

```
Limit     : constant Integer := 10_000;
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Real := Dispersion(1.15);
Hello_Msg : constant access String := Hello'Access; -- see 3.10.2
```

3.3.2 Number Declarations

Replace paragraph 10:

```
Max           : constant := 500;                -- an integer number
Max_Line_Size : constant := Max/6               -- the integer 83
Power_16      : constant := 2**16;              -- the integer 65_536
One, Un, Eins : constant := 1;                  -- three different names for 1
```

by:

```
Max           : constant := 500;                -- an integer number
Max_Line_Size : constant := Max/6;              -- the integer 83
Power_16      : constant := 2**16;              -- the integer 65_536
One, Un, Eins : constant := 1;                  -- three different names for 1
```

3.4 Derived Types and Classes

Replace paragraph 1:

A derived_type_definition defines a new type (and its first subtype) whose characteristics are *derived* from those of a *parent type*.

by:

A derived_type_definition defines a *derived type* (and its first subtype) whose characteristics are *derived* from those of a parent type, and possibly from progenitor types.

A *class of types* is a set of types that is closed under derivation; that is, if the parent or a progenitor type of a derived type belongs to a class, then so does the derived type. By saying that a particular group of types forms a class, we are saying that all derivatives of a type in the set inherit the characteristics that define that set. The more general term *category of types* is used for a set of types whose defining characteristics are not necessarily inherited by derivatives; for example, limited, abstract, and interface are all categories of types, but not classes of types.

Replace paragraph 2:

derived_type_definition ::= [**abstract**] **new** *parent*_subtype_indication [record_extension_part]

by:

derived_type_definition ::=
[**abstract**] [**limited**] **new** *parent*_subtype_indication [[**and** interface_list] record_extension_part]

Replace paragraph 3:

The *parent*_subtype_indication defines the *parent subtype*; its type is the parent type.

by:

The *parent*_subtype_indication defines the *parent subtype*; its type is the *parent type*. The interface_list defines the progenitor types (see 3.9.4). A derived type has one parent type and zero or more progenitor types.

Replace paragraph 5:

If there is a record_extension_part, the derived type is called a *record extension* of the parent type. A record_extension_part shall be provided if and only if the parent type is a tagged type.

by:

If there is a record_extension_part, the derived type is called a *record extension* of the parent type. A record_extension_part shall be provided if and only if the parent type is a tagged type. An interface_list shall be provided only if the parent type is a tagged type.

If the reserved word **limited** appears in a derived_type_definition, the parent type shall be a limited type.

Insert after paragraph 6:

The first subtype of the derived type is unconstrained if a known_discriminant_part is provided in the declaration of the derived type, or if the parent subtype is unconstrained. Otherwise, the constraint of the first subtype *corresponds* to that of the parent subtype in the following sense: it is the same as that of the parent subtype except that for a range constraint (implicit or explicit), the value of each bound of its range is replaced by the corresponding value of the derived type.

the new paragraph:

The first subtype of the derived type excludes null (see 3.10) if and only if the parent subtype excludes null.

Replace paragraph 8:

- Each class of types that includes the parent type also includes the derived type.

by:

- If the parent type or a progenitor type belongs to a class of types, then the derived type also belongs to that class. The following sets of types, as well as any higher-level sets composed from them, are classes in this sense, and hence the characteristics defining these classes are inherited by derived types from their parent or progenitor types: signed integer, modular integer, ordinary fixed, decimal fixed, floating point, enumeration, boolean, character, access-to-constant, general access-to-variable, pool-specific access-to-variable, access-to-subprogram, array, string, non-array composite, nonlimited, untagged record, tagged, task, protected, and synchronized tagged.

Delete paragraph 15:

- The derived type is limited if and only if the parent type is limited.

Replace paragraph 17:

- For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type that already exists at the place of the derived_type_definition, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is

rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2).

by:

- For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type or of a progenitor type that already exists at the place of the derived_type_definition, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type and any progenitor types are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2).

Replace paragraph 18:

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent type with a *corresponding subtype* of the derived type. For a given subtype of the parent type, the corresponding subtype of the derived type is defined as follows:

by:

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent or progenitor type with a *corresponding subtype* of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

Replace paragraph 22:

The same formal parameters have default_expressions in the profile of the inherited subprogram. Any type mismatch due to the systematic replacement of the parent type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.

by:

The same formal parameters have default_expressions in the profile of the inherited subprogram. Any type mismatch due to the systematic replacement of the parent or progenitor type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.

Replace paragraph 23:

If a primitive subprogram of the parent type is visible at the place of the derived_type_definition, then the corresponding inherited subprogram is implicitly declared immediately after the derived_type_definition. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

by:

If a primitive subprogram of the parent or progenitor type is visible at the place of the derived_type_definition, then the corresponding inherited subprogram is implicitly declared immediately after the derived_type_definition. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

Replace paragraph 27:

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the parent's subprogram is converted to the derived type.

by:

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent or progenitor type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the subprogram of the parent or progenitor is converted to the derived type, or in the

case of a null extension, extended to the derived type using the equivalent of an extension_aggregate with the original result as the ancestor_part and **null record** as the record_component_association_list.

Insert after paragraph 35:

17 If the reserved word **abstract** is given in the declaration of a type, the type is abstract (see 3.9.3).

the new paragraphs:

18 An interface type that has a progenitor type "is derived from" that type. A derived_type_definition, however, never defines an interface type.

19 It is illegal for the parent type of a derived_type_definition to be a synchronized tagged type.

3.4.1 Derivation Classes

Replace paragraph 2:

A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below).

by:

A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. A derived type, interface type, type extension, task type, protected type, or formal derived type is also derived from every ancestor of each of its progenitor types, if any. The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below).

Replace paragraph 3:

Every type is either a *specific* type, a *class-wide* type, or a *universal* type. A specific type is one defined by a type_declaration, a formal_type_declaration, or a full type definition embedded in a declaration for an object. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

by:

Every type is either a *specific* type, a *class-wide* type, or a *universal* type. A specific type is one defined by a type_declaration, a formal_type_declaration, or a full type definition embedded in another construct. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

Replace paragraph 6:

Universal types

Universal types are defined for (and belong to) the integer, real, and fixed point classes, and are referred to in this standard as respectively, *universal_integer*, *universal_real*, and *universal_fixed*. These are analogous to class-wide types for these language-defined numeric classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real numeric_literal) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

by:

Universal types

Universal types are defined for (and belong to) the integer, real, fixed point, and access classes, and are referred to in this standard as respectively, *universal_integer*, *universal_real*, *universal_fixed*, and *universal_access*. These are analogous to class-wide types for these language-defined elementary classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real numeric_literal, or the literal **null**) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

Replace paragraph 10:

A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. The *ultimate ancestor* of a type is the ancestor of the type that is not a descendant of any other type.

by:

A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the numeric universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. An *ultimate ancestor* of a type is an ancestor of that type that is not itself a descendant of any other type. Every untagged type has a unique ultimate ancestor.

3.5 Scalar Types

Insert after paragraph 27:

For an enumeration type, the function returns the value whose position number is one less than that of the value of Arg; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of Arg. For a fixed point type, the function returns the result of subtracting small from the value of Arg. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of Arg; Constraint_Error is raised if there is no such machine number.

the new paragraphs:

S'Wide_Wide_Image

S'Wide_Wide_Image denotes a function with the following specification:

```
function S'Wide_Wide_Image(Arg : S'Base)
  return Wide_Wide_String
```

The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. The lower bound of the result is one.

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 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. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the result is a corresponding language-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is "NUL" — the quotes are not part of the image).

The image of a floating point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits–1 (see 3.5.8) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or –), and two or more digits (with leading zeros if necessary) representing the exponent. If S'Signed_Zeros is True, then the leading character is a minus sign for a negatively signed zero.

The image of a fixed point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see 3.5.10) digits after the decimal point.

Replace paragraph 30:

The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. The lower bound of the result is one.

by:

The function returns an image of the value of *Arg* as a Wide_String. The lower bound of the result is one. The image has the same sequence of character as defined for S'Wide_Wide_Image if all the graphic characters are defined in Wide_Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Wide_Image for the same value of Arg).

Delete paragraph 31:

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.

Delete paragraph 32:

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. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the result is a corresponding language-defined or implementation-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is "NUL" — the quotes are not part of the image).

Delete paragraph 33:

The image of a floating point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits-1 (see 3.5.8) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or –), and two or more digits (with leading zeros if necessary) representing the exponent. If S'Signed_Zeros is True, then the leading character is a minus sign for a negatively signed zero.

Delete paragraph 34:

The image of a fixed point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see 3.5.10) digits after the decimal point.

Replace paragraph 37:

The function returns an image of the value of *Arg* as a String. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for S'Wide_Image if all the graphic characters are defined in Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Image for the same value of *Arg*).

by:

The function returns an image of the value of *Arg* as a String. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for S'Wide_Wide_Image if all the graphic characters are defined in Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Wide_Image for the same value of *Arg*).

S'Wide_Wide_Width

S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

Insert after paragraph 39:

S'Width

S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

the new paragraphs:

S'Wide_Wide_Value

S'Wide_Wide_Value denotes a function with the following specification:

```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
  return S'Base
```

This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces.

For the evaluation of a call on S'Wide_Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Wide_Image for a nongraphic character of the type), the result is the corresponding enumeration value; otherwise Constraint_Error is raised.

For the evaluation of a call on S'Wide_Wide_Value for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint_Error is raised.

For the evaluation of a call on S'Wide_Wide_Value for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

- numeric_literal
- numeral.[exponent]
- .numeral[exponent]
- base#based_numeral.#[exponent]
- base#.based_numeral#[exponent]

with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed_Zeros is True.

Replace paragraph 43:

For the evaluation of a call on S'Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Image for a nongraphic character of the type), the result is the corresponding enumeration value; otherwise Constraint_Error is raised.

by:

For the evaluation of a call on S'Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Image for a value of the type), the result is the corresponding enumeration value; otherwise Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Wide_Value with *Arg* of type Wide_String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String.

Delete paragraph 44:

For the evaluation of a call on S'Wide_Value (or S'Value) for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint_Error is raised.

Delete paragraph 45:

For the evaluation of a call on S'Wide_Value (or S'Value) for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

Delete paragraph 46:

- numeric_literal

Delete paragraph 47:

- numeral.[exponent]

Delete paragraph 48:

- .numeral[exponent]

Delete paragraph 49:

- base#based_numeral.#[exponent]

Delete paragraph 50:

- base#.based_numeral#[exponent]

Delete paragraph 51:

with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed_Zeros is True.

Replace paragraph 55:

For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type), the result is the corresponding enumeration value; otherwise Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with *Arg* of type String is equivalent to a call on S'Wide_Value for a corresponding *Arg* of type Wide_String.

by:

For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type), the result is the corresponding enumeration value; otherwise Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with *Arg* of type String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String.

Replace paragraph 56:

An implementation may extend the Wide_Value, Value, Wide_Image, and Image attributes of a floating point type to support special values such as infinities and NaNs.

by:

An implementation may extend the Wide_Wide_Value, Wide_Value, Value, Wide_Wide_Image, Wide_Image, and Image attributes of a floating point type to support special values such as infinities and NaNs.

Replace paragraph 59:

21 For any value V (including any nongraphic character) of an enumeration subtype S, S'Value(S'Image(V)) equals V, as does S'Wide_Value(S'Wide_Image(V)). Neither expression ever raises Constraint_Error.

by:

21 For any value V (including any nongraphic character) of an enumeration subtype S, S'Value(S'Image(V)) equals V, as do S'Wide_Value(S'Wide_Image(V)) and S'Wide_Wide_Value(S'Wide_Wide_Image(V)). None of these expressions ever raise Constraint_Error.

3.5.2 Character Types

Replace paragraph 2:

The predefined type Character is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO 10646 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding character_literal in Character. Each of the nongraphic positions of Row 00 (0000-001F and 007F-009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes (Wide_)Image and (Wide_)Value; these names are given in the definition of type Character in A.1, "The Package Standard", but are set in *italics*.

by:

The predefined type Character is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding character_literal in Character. Each of the nongraphic positions of Row 00 (0000-001F and 007F-009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes Image, Wide_Image, Wide_Wide_Image, Value, Wide_Value, and Wide_Wide_Value; these names are given in the definition of type Character in A.1, "The Package Standard", but are set in *italics*.

Replace paragraph 3:

The predefined type Wide_Character is a character type whose values correspond to the 65536 code positions of the ISO 10646 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding character_literal in Wide_Character. The first 256 values of Wide_Character have the same character_literal or language-defined name as defined for Character. The last 2 values of Wide_Character correspond to the nongraphic positions FFFE and FFFF of the BMP, and are assigned the language-defined names *FFFE* and *FFFF*. As with the other language-defined names for nongraphic characters, the names *FFFE* and *FFFF* are usable only with the attributes (Wide_)Image and (Wide_)Value; they are not usable as enumeration literals. All other values of Wide_Character are considered graphic characters, and have a corresponding character_literal.

by:

The predefined type Wide_Character is a character type whose values correspond to the 65536 code positions of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding character_literal in Wide_Character. The first 256 values of Wide_Character have the same character_literal or language-defined name as defined for Character. Each of the graphic_characters has a corresponding character_literal.

The predefined type Wide_Wide_Character is a character type whose values correspond to the 2147483648 code positions of the ISO/IEC 10646:2003 character set. Each of the graphic_characters has a corresponding character_literal in Wide_Wide_Character. The first 65536 values of Wide_Wide_Character have the same character_literal or language-defined name as defined for Wide_Character.

The characters whose code position is larger than 16#FF# and which are not graphic_characters have language-defined names which are formed by appending to the string "Hex_" the representation of their code position in hexadecimal as eight extended digits. As with other language-defined names, these names are usable only with the attributes (Wide_)Wide_Image and (Wide_)Wide_Value; they are not usable as enumeration literals.

Delete paragraph 4:

In a nonstandard mode, an implementation may provide other interpretations for the predefined types Character and Wide_Character, to conform to local conventions.

Delete paragraph 5:

If an implementation supports a mode with alternative interpretations for Character and Wide_Character, the set of graphic characters of Character should nevertheless remain a proper subset of the set of graphic characters of Wide_Character. Any character set "localizations" should be reflected in the results of the subprograms defined in the language-defined package Characters.Handling (see A.3) available in such a mode. In a mode with an alternative interpretation of Character, the implementation should also support a corresponding change in what is a legal identifier_letter.

3.5.4 Integer Types

Replace paragraph 16:

For every modular subtype S, the following attribute is defined:

by:

For every modular subtype S, the following attributes are defined:

S'Mod

S'Mod denotes a function with the following specification:

```
function S'Mod (Arg : universal_integer)
     return S'Base
```

This function returns *Arg* **mod** S'Modulus, as a value of the type of S.

3.5.9 Fixed Point Types

Replace paragraph 8:

The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. For a type defined by an ordinary_fixed_point_definition (an *ordinary* fixed point type), the *small* may be specified by an attribute_definition_clause (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*.

by:

The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. The *machine numbers* of a fixed point type are the values of the type that can be represented exactly in every unconstrained variable of the type. For a type defined by an ordinary_fixed_point_definition (an *ordinary* fixed point type), the *small* may be specified by an attribute_definition_clause (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*.

3.6 Array Types

Replace paragraph 7:

component_definition ::= [**aliased**] subtype_indication

by:

component_definition ::=
  [**aliased**] subtype_indication
 | [**aliased**] access_definition

Delete paragraph 11:

Within the definition of a nonlimited composite type (or a limited composite type that later in its immediate scope becomes nonlimited — see 7.3.1 and 7.5), if a component_definition contains the reserved word **aliased** and the type of the component is discriminated, then the nominal subtype of the component shall be constrained.

Replace paragraph 22:

The elaboration of a discrete_subtype_definition that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a discrete_subtype_definition that contains one or more per-object expressions is defined in 3.8. The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

by:

The elaboration of a discrete_subtype_definition that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a discrete_subtype_definition that contains one or more per-object expressions is defined in 3.8. The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication or access_definition. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

Replace paragraph 30:

```
Grid : array(1 .. 80, 1 .. 100) of Boolean;
Mix  : array(Color range Red .. Green) of Boolean;
Page : array(Positive range <>) of Line :=  -- an array of arrays
  (1 | 50  => Line'(1 | Line'Last => '+', others => '-'),  -- see 4.3.3
   2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
    -- Page is constrained by its initial value to (1..50)
```

by:

```
Grid      : array(1 .. 80, 1 .. 100) of Boolean;
Mix       : array(Color range Red .. Green) of Boolean;
Msg_Table : constant array(Error_Code) of access constant String :=
      (Too_Big => new String'("Result too big"), Too_Small => ...);
Page      : array(Positive range <>) of Line :=  -- an array of arrays
  (1 | 50  => Line'(1 | Line'Last => '+', others => '-'),  -- see 4.3.3
   2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
    -- Page is constrained by its initial value to (1..50)
```

3.6.2 Operations of Array Types

Replace paragraph 16:

48 A component of an array can be named with an indexed_component. A value of an array type can be specified with an array_aggregate, unless the array type is limited. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type.

by:

48 A component of an array can be named with an indexed_component. A value of an array type can be specified with an array_aggregate. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type.

3.6.3 String Types

Replace paragraph 2:

There are two predefined string types, String and Wide_String, each indexed by values of the predefined subtype Positive; these are declared in the visible part of package Standard:

by:

There are three predefined string types, String, Wide_String, and Wide_Wide_String, each indexed by values of the predefined subtype Positive; these are declared in the visible part of package Standard:

Replace paragraph 4:

```
type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
```

by:

```
type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
```

3.7 Discriminants

Replace paragraph 1:

A composite type (other than an array type) can have discriminants, which parameterize the type. A known_discriminant_part specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An unknown_discriminant_part in the declaration of a partial view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a partial view are indefinite subtypes.

by:

A composite type (other than an array or interface type) can have discriminants, which parameterize the type. A known_discriminant_part specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An unknown_discriminant_part in the declaration of a view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a view are indefinite subtypes.

Replace paragraph 5:

```
discriminant_specification ::=
   defining_identifier_list : subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]
```

by:

```
discriminant_specification ::=
   defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]
```

Replace paragraph 8:

A discriminant_part is only permitted in a declaration for a composite type that is not an array type (this includes generic formal types). A type declared with a known_discriminant_part is called a *discriminated* type, as is a type that inherits (known) discriminants.

by:

A discriminant_part is only permitted in a declaration for a composite type that is not an array or interface type (this includes generic formal types). A type declared with a known_discriminant_part is called a *discriminated* type, as is a type that inherits (known) discriminants.

Replace paragraph 9:

The subtype of a discriminant may be defined by a subtype_mark, in which case the subtype_mark shall denote a discrete or access subtype, or it may be defined by an access_definition (in which case the subtype_mark of the access_definition may denote any kind of subtype). A discriminant that is defined by an access_definition is called an *access discriminant* and is of an anonymous general access-to-variable type whose designated subtype is denoted by the subtype_mark of the access_definition.

by:

The subtype of a discriminant may be defined by an optional null_exclusion and a subtype_mark, in which case the subtype_mark shall denote a discrete or access subtype, or it may be defined by an access_definition. A discriminant that is defined by an access_definition is called an *access discriminant* and is of an anonymous access type.

Replace paragraph 10:

A discriminant_specification for an access discriminant shall appear only in the declaration for a task or protected type, or for a type with the reserved word **limited** in its (full) definition or in that of one of its ancestors. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

by:

Default_expressions shall be provided either for all or for none of the discriminants of a known_discriminant_part. No default_expressions are permitted in a known_discriminant_part in a declaration of a tagged type or a generic formal type.

A discriminant_specification for an access discriminant may have a default_expression only in the declaration for a task or protected type, or for a type that is a descendant of an explicitly limited record type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Delete paragraph 11:

Default_expressions shall be provided either for all or for none of the discriminants of a known_discriminant_part. No default_expressions are permitted in a known_discriminant_part in a declaration of a tagged type [or a generic formal type].

Replace paragraph 27:

An access_definition is elaborated when the value of a corresponding access discriminant is defined, either by evaluation of its default_expression or by elaboration of a discriminant_constraint. The elaboration of an access_definition creates the anonymous access type. When the expression defining the access discriminant is evaluated, it is converted to this anonymous access type (see 4.6).

by:

For an access discriminant, its access_definition is elaborated when the value of the access discriminant is defined: by evaluation of its default_expression, by elaboration of a discriminant_constraint, or by an assignment that initializes the enclosing object.

Replace paragraph 37:

```
type Item(Number : Positive) is
   record
      Content : Integer;
      -- no component depends on the discriminant
   end record;
```

by:

```
task type Worker(Prio : System.Priority; Buf : access Buffer) is
   -- discriminants used to parameterize the task type (see 9.1)
   pragma Priority(Prio);  -- see D.1
   entry Fill;
   entry Drain;
end Worker;
```

3.7.1 Discriminant Constraints

Replace paragraph 7:

A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of a general access subtype, a discriminant_constraint is illegal if there is a place within the immediate scope of the designated subtype where the designated subtype's view is constrained.

by:

A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant_constraint is illegal if the designated type has a partial view that is constrained or, for a general access subtype, has default_expressions for its discriminants. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. In a generic body, this rule is checked presuming all formal access types of the generic might be general access types, and all untagged discriminated formal types of the generic might have default_expressions for their discriminants.

3.8 Record Types

Delete paragraph 8:

A default_expression is not permitted if the component is of a limited type.

Replace paragraph 9:

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 shall be distinct.

by:

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 shall be distinct.

Insert before paragraph 14:

The component_definition of a component_declaration defines the (nominal) subtype of the component. If the reserved word **aliased** appears in the component_definition, then the component is aliased (see 3.10).

the new paragraph:

If a record_type_declaration includes the reserved word **limited**, the type is called an *explicitly limited record* type.

Replace paragraph 18:

Within the definition of a composite type, if a component_definition or discrete_subtype_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a component_definition of a component_declaration or the discrete_subtype_definition of an entry_declaration for an entry family (see 9.5.2), if the constraint or range of the subtype_indication or discrete_subtype_definition is not a per-object constraint, then the subtype_indication or discrete_subtype_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

by:

Within the definition of a composite type, if a component_definition or discrete_subtype_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a component_definition of a component_declaration or the discrete_subtype_definition of an entry_declaration for an entry family (see 9.5.2), if the component subtype is defined by an access_definition or if the constraint or range of the subtype_indication or discrete_subtype_definition is not a per-object constraint, then the access_definition, subtype_indication, or discrete_subtype_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

Replace paragraph 25:

61 A component of a record can be named with a selected_component. A value of a record can be specified with a record_aggregate, unless the record type is limited.

by:

61 A component of a record can be named with a selected_component. A value of a record can be specified with a record_aggregate.

3.9 Tagged Types and Type Extensions

Replace paragraph 2:

A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. When deriving from a tagged type, additional components may be defined. As for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden. The derived type is called an *extension* of the ancestor type, or simply a *type extension*. Every type extension is also a tagged type, and is either a *record extension* or a *private extension* of some other tagged type. A record extension is defined by a derived_type_definition with a record_extension_part. A private extension, which is a partial view of a record extension, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1).

by:

A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. In addition, an interface type is a tagged type, as is a task or protected type derived from an interface (see 3.9.4). When deriving from a tagged type, as for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden. The derived type is called an *extension* of its ancestor types, or simply a *type extension*.

Every type extension is also a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a non-interface synchronized tagged type (see 3.9.4). A record extension is defined by a derived_type_definition with a record_extension_part (see 3.9.1), which may include the definition of additional components. A private extension, which is a partial view of a record extension or of a synchronized tagged type, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1).

Replace paragraph 4:

The tag of a specific tagged type identifies the full_type_declaration of the type. If a declaration for a tagged type occurs within a generic_package_declaration, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.

by:

The tag of a specific tagged type identifies the full_type_declaration of the type, and for a type extension, is sufficient to uniquely identify the type among all descendants of the same ancestor. If a declaration for a tagged type occurs within a generic_package_declaration, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body and with all of its ancestors (if any) also local to the generic body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.

Replace paragraph 6:

```
package Ada.Tags is
    type Tag is private;
```

by:

```
package Ada.Tags is
    pragma Preelaborate(Tags);
    type Tag is private;

    No_Tag : constant Tag;
```

Insert after paragraph 7:

```
    function Expanded_Name(T : Tag) return String;
    function External_Tag(T : Tag) return String;
    function Internal_Tag(External : String) return Tag;
```

the new paragraphs:

```
    function Descendant_Tag(External : String; Ancestor : Tag) return Tag;
    function Is_Descendant_At_Same_Level(Descendant, Ancestor : Tag)
        return Boolean;

    function Parent_Tag (T : Tag) return Tag;
```

```
type Tag_Array is array (Positive range <>) of Tag;
function Interface_Ancestor_Tags (T : Tag) return Tag_Array;
```

Insert after paragraph 9:

```
private
   ... -- not specified by the language
end Ada.Tags;
```

the new paragraph:

No_Tag is the default initial value of type Tag.

Replace paragraph 10:

The function Expanded_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed block_statement.

by:

The function Wide_Wide_Expanded_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed block_statement.

The function Expanded_Name (respectively, Wide_Expanded_Name) returns the same sequence of graphic characters as that defined for Wide_Wide_Expanded_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Expanded_Name for the same value of the argument.

Insert after paragraph 11:

The function External_Tag returns a string to be used in an external representation for the given tag. The call External_Tag(S'Tag) is equivalent to the attribute_reference S'External_Tag (see 13.3).

the new paragraph:

The string returned by the functions Expanded_Name, Wide_Expanded_Name, Wide_Wide_Expanded_Name, and External_Tag has lower bound 1.

Replace paragraph 12:

The function Internal_Tag returns the tag that corresponds to the given external tag, or raises Tag_Error if the given string is not the external tag for any specific type of the partition.

by:

The function Internal_Tag returns a tag that corresponds to the given external tag, or raises Tag_Error if the given string is not the external tag for any specific type of the partition. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created (see 13.14).

The function Descendant_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag_Error is raised if External is not the external tag for such a type. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created.

The function Is_Descendant_At_Same_Level returns True if the Descendant tag identifies a type that is both a descendant of the type identified by Ancestor and at the same accessibility level. If not, it returns False.

The function Parent_Tag returns the tag of the parent type of the type whose tag is T. If the type does not have a parent type (that is, it was not declared by a derived_type_declaration), then No_Tag is returned.

The function Interface_Ancestor_Tags returns an array containing the tag of each interface ancestor type of the type whose tag is T, other than T itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once. If the type whose tag is T has no interface ancestors, a null array is returned.

Insert after paragraph 18:

X'Tag

X'Tag denotes the tag of X. The value of this attribute is of type Tag.

the new paragraphs:

The following language-defined generic function exists:

```
generic
    type T (<>) is abstract tagged limited private;
    type Parameters (<>) is limited private;
    with function Constructor (Params : not null access Parameters)
        return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
   (The_Tag : Tag;
    Params  : not null access Parameters) return T'Class;
pragma Preelaborate(Generic_Dispatching_Constructor);
pragma Convention(Intrinsic, Generic_Dispatching_Constructor);
```

Tags.Generic_Dispatching_Constructor provides a mechanism to create an object of an appropriate type from just a tag value. The function Constructor is expected to create the object given a reference to an object of type Parameters.

Replace paragraph 24:

- The tag of the result returned by a function with a class-wide result type is that of the return expression.

by:

- The tag of the result returned by a function with a class-wide result type is that of the return object.

Insert after paragraph 25:

The tag is preserved by type conversion and by parameter passing. The tag of a value is the tag of the associated object (see 6.2).

the new paragraphs:

Tag_Error is raised by a call of Descendant_Tag, Expanded_Name, External_Tag, Interface_Ancestor_Tag, Is_Descendant_At_Same_Level, or Parent_Tag if any tag passed is No_Tag.

An instance of Tags.Generic_Dispatching_Constructor raises Tag_Error if The_Tag does not represent a concrete descendant of T or if the innermost master (see 7.6.1) of this descendant is not also a master of the instance. Otherwise, it dispatches to the primitive function denoted by the formal Constructor for the type identified by The_Tag, passing Params, and returns the result. Any exception raised by the function is propagated.

Erroneous Execution

If an internal tag provided to an instance of Tags.Generic_Dispatching_Constructor or to any subprogram declared in package Tags identifies either a type that is not library-level and whose tag has not been created (see 13.14), or a type that does not exist in the partition at the time of the call, then execution is erroneous.

Replace paragraph 26:

The implementation of the functions in Ada.Tags may raise Tag_Error if no specific type corresponding to the tag passed as a parameter exists in the partition at the time the function is called.

by:

The implementation of Internal_Tag and Descendant_Tag may raise Tag_Error if no specific type corresponding to the string External passed as a parameter exists in the partition at the time the function is called, or if there is no such type whose innermost master is a master of the point of the function call.

Implementation Advice

Internal_Tag should return the tag of a type whose innermost master is the master of the point of the function call.

Replace paragraph 30:

65 If S denotes an untagged private type whose full type is tagged, then S'Class is also allowed before the full type definition, but only in the private part of the package in which the type is declared (see 7.3.1). Similarly, the Class attribute is defined for incomplete types whose full type is tagged, but only within the library unit in which the incomplete type is declared (see 3.10.1).

by:

65 The capability provided by Tags.Generic_Dispatching_Constructor is sometimes known as a *factory*.

3.9.1 Type Extensions

Replace paragraph 1:

Every type extension is a tagged type, and is either a *record extension* or a *private extension* of some other tagged type.

by:

Every type extension is a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a non-interface synchronized tagged type.

Replace paragraph 3:

The parent type of a record extension shall not be a class-wide type. If the parent type is nonlimited, then each of the components of the record_extension_part shall be nonlimited. The accessibility level (see 3.10.2) of a record extension shall not be statically deeper than that of its parent type. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

by:

The parent type of a record extension shall not be a class-wide type nor shall it be a synchronized tagged type (see 3.9.4). If the parent type or any progenitor is nonlimited, then each of the components of the record_extension_part shall be nonlimited. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

Replace paragraph 4:

A type extension shall not be declared in a generic body if the parent type is declared outside that body.

by:

Within the body of a generic unit, or the body of any of its descendant library units, a tagged type shall not be declared as a descendant of a formal type declared within the formal part of the generic unit.

Static Semantics

A record extension is a *null extension* if its declaration has no known_discriminant_part and its record_extension_part includes no component_declarations.

Replace paragraph 7:

The accessibility rules imply that a tagged type declared in a library package_specification can be extended only at library level or as a generic formal. When the extension is declared immediately within a package_body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

by:

When an extension is declared immediately within a body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

3.9.2 Dispatching Operations of Tagged Types

Replace paragraph 1:

The primitive subprograms of a tagged type are called *dispatching operations*. A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time.

Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time; such a call is termed a *dispatching call*. As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.

by:

The primitive subprograms of a tagged type, the subprograms declared by formal_abstract_subprogram_declarations, and the stream attributes of a specific tagged type that are available (see 13.13.2) at the end of the declaration list where the type is declared are called *dispatching operations*. A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time; such a call is termed a *dispatching call*. As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.

Replace paragraph 2:

A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a primitive subprogram of a tagged type, that is, a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T*, then the call has a *controlling result* — the context of the call can control the dispatching.

by:

A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T*, then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with access result type designating *T*, then the call has a *controlling access result*, and the context can similarly control dispatching.

Replace paragraph 4:

- The name or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result, it has at least one statically tagged controlling operand;

by:

- The name or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result or controlling access result, it has at least one statically tagged controlling operand;

Replace paragraph 5:

- The name or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result and at least one dynamically tagged controlling operand;

by:

- The name or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result or controlling access result and at least one dynamically tagged controlling operand;

Replace paragraph 6:

- The name or expression is *tag indeterminate* if it is a call with a controlling result, all of whose controlling operands (if any) are tag indeterminate.

by:

- The name or expression is *tag indeterminate* if it is a call with a controlling result or controlling access result, all of whose controlling operands (if any) are tag indeterminate.

Replace paragraph 10:

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited or overriding dispatching operation is the convention of the corresponding primitive operation of the parent type. An explicitly declared dispatching operation shall not be of convention Intrinsic.

by:

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited dispatching operation is the convention of the corresponding primitive operation of the parent or progenitor type. The default convention of a dispatching operation that overrides an inherited primitive operation is the convention of the inherited operation; if the operation overrides multiple inherited operations, then they shall all have the same convention. An explicitly declared dispatching operation shall not be of convention Intrinsic.

Replace paragraph 11:

The default_expression for a controlling formal parameter of a dispatching operation shall be tag indeterminate. A controlling formal parameter that is an access parameter shall not have a default_expression.

by:

The default_expression for a controlling formal parameter of a dispatching operation shall be tag indeterminate.

If a dispatching operation is defined by a subprogram_renaming_declaration or the instantiation of a generic subprogram, any access parameter of the renamed subprogram or the generic subprogram that corresponds to a controlling access parameter of the dispatching operation, shall have a subtype that excludes null.

Replace paragraph 17:

If all of the controlling operands are tag-indeterminate, then:

by:

If all of the controlling operands (if any) are tag-indeterminate, then:

Replace paragraph 18:

- If the call has a controlling result and is itself a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of type *T*, then its controlling tag value is determined by the controlling tag value of this enclosing call;

by:

- If the call has a controlling result or controlling access result and is itself, or designates, a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of a descendant of type *T*, then its controlling tag value is determined by the controlling tag value of this enclosing call;
- If the call has a controlling result or controlling access result and (possibly parenthesized, qualified, or dereferenced) is the expression of an assignment_statement whose target is of a class-wide type, then its controlling tag value is determined by the target;

Replace paragraph 20:

For the execution of a call on a dispatching operation, the body executed is the one for the corresponding primitive subprogram of the specific type identified by the controlling tag value. The body for an explicitly declared dispatching operation is the corresponding explicit body for the subprogram. The body for an implicitly declared dispatching operation that is overridden is the body for the overriding subprogram, even if the overriding occurs in a private part. The

body for an inherited dispatching operation that is not overridden is the body of the corresponding subprogram of the parent or ancestor type.

by:

For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value. If the corresponding operation is explicitly declared for this type, even if the declaration occurs in a private part, then the action comprises an invocation of the explicit body for the operation. If the corresponding operation is implicitly declared for this type:

- if the operation is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any;
- otherwise, the action is the same as the action for the corresponding operation of the parent type.

Replace paragraph 22:

73 This subclause covers calls on primitive subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an assignment_statement is described in 5.2.

by:

73 This subclause covers calls on dispatching subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an assignment_statement is described in 5.2.

3.9.3 Abstract Types and Subprograms

Replace paragraph 1:

An *abstract type* is a tagged type intended for use as a parent type for type extensions, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.

by:

An *abstract type* is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.

Syntax

abstract_subprogram_declaration ::=
[overriding_indicator]
subprogram_specification **is abstract**;

Static Semantics

Interface types (see 3.9.4) are abstract types. In addition, a tagged type that has the reserved word **abstract** in its declaration is an abstract type. The class-wide type (see 3.4.1) rooted at an abstract type is not itself an abstract type.

Replace paragraph 2:

An abstract type is a specific type that has the reserved word abstract in its declaration. Only a tagged type is allowed to be declared abstract.

by:

Only a tagged type shall have the reserved word **abstract** in its declaration.

Replace paragraph 3:

A subprogram declared by an abstract_subprogram_declaration (see 6.1) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be abstract.

by:

A subprogram declared by an abstract_subprogram_declaration or a formal_abstract_subprogram_declaration (see 12.6) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be abstract.

Replace paragraph 4:

For a derived type, if the parent or ancestor type has an abstract primitive subprogram, or a primitive function with a controlling result, then:

by:

If a type has an implicitly declared primitive subprogram that is inherited or is the predefined equality operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a null extension inherits a function with a controlling result, then:

Replace paragraph 5:

- If the derived type is abstract or untagged, the inherited subprogram is *abstract*.

by:

- If the type is abstract or untagged, the implicitly declared subprogram is *abstract*.

Replace paragraph 6:

Otherwise, the subprogram shall be overridden with a nonabstract subprogram; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

by:

Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a function with a controlling result, have a full type that is a null extension; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. Such a subprogram is said to *require overriding*. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

Replace paragraph 11:

A generic actual subprogram shall not be an abstract subprogram. The prefix of an attribute_reference for the Access, Unchecked_Access, or Address attributes shall not denote an abstract subprogram.

by:

A generic actual subprogram shall not be an abstract subprogram unless the generic formal subprogram is declared by a formal_abstract_subprogram_declaration. The prefix of an attribute_reference for the Access, Unchecked_Access, or Address attributes shall not denote an abstract subprogram.

Dynamic Semantics

The elaboration of an abstract_subprogram_declaration has no effect.

3.9.4 Interface Types

Insert new clause:

An interface type is an abstract tagged type that provides a restricted form of multiple inheritance. A tagged type, task type, or protected type may have one or more interface types as ancestors.

Syntax

interface_type_definition ::=
  [**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list]

interface_list ::= *interface*_subtype_mark {**and** *interface*_subtype_mark}

Static Semantics

An interface type (also called an *interface*) is a specific abstract tagged type that is defined by an interface_type_definition.

An interface with the reserved word **limited**, **task**, **protected**, or **synchronized** in its definition is termed, respectively, a *limited interface*, a *task interface*, a *protected interface*, or a *synchronized interface*. In addition, all task and protected interfaces are synchronized interfaces, and all synchronized interfaces are limited interfaces.

A task or protected type derived from an interface is a tagged type. Such a tagged type is called a *synchronized* tagged type, as are synchronized interfaces and private extensions whose declaration includes the reserved word **synchronized**.

A task interface is an abstract task type. A protected interface is an abstract protected type.

An interface type has no components.

An *interface*_subtype_mark in an interface_list names a *progenitor subtype*; its type is the *progenitor type*. An interface type inherits user-defined primitive subprograms from each progenitor type in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4).

Legality Rules

All user-defined primitive subprograms of an interface type shall be abstract subprograms or null procedures.

The type of a subtype named in an interface_list shall be an interface type.

A type derived from a nonlimited interface shall be nonlimited.

An interface derived from a task interface shall include the reserved word **task** in its definition; any other type derived from a task interface shall be a private extension or a task type declared by a task declaration (see 9.1).

An interface derived from a protected interface shall include the reserved word **protected** in its definition; any other type derived from a protected interface shall be a private extension or a protected type declared by a protected declaration (see 9.4).

An interface derived from a synchronized interface shall include one of the reserved words **task**, **protected**, or **synchronized** in its definition; any other type derived from a synchronized interface shall be a private extension, a task type declared by a task declaration, or a protected type declared by a protected declaration.

No type shall be derived from both a task interface and a protected interface.

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

Dynamic Semantics

The elaboration of an interface_type_definition has no effect.

NOTES

79 Nonlimited interface types have predefined nonabstract equality operators. These may be overridden with user-defined abstract equality operators. Such operators will then require an explicit overriding for any nonabstract descendant of the interface.

Examples

Example of a limited interface and a synchronized interface extending it:

```
type Queue is limited interface;
procedure Append(Q : in out Queue; Person : in Person_Name) is abstract;
procedure Remove_First(Q      : in out Queue;
                       Person : out Person_Name) is abstract;
function Cur_Count(Q : in Queue) return Natural is abstract;
function Max_Count(Q : in Queue) return Natural is abstract;
-- See 3.10.1 for Person_Name.

Queue_Error : exception;
-- Append raises Queue_Error if Count(Q) = Max_Count(Q)
-- Remove_First raises Queue_Error if Count(Q) = 0
```

```
type Synchronized_Queue is synchronized interface and Queue; -- see 9.11
procedure Append_Wait(Q      : in out Synchronized_Queue;
                      Person : in Person_Name) is abstract;
procedure Remove_First_Wait(Q      : in out Synchronized_Queue;
                            Person : out Person_Name) is abstract;

...

procedure Transfer(From   : in out Queue'Class;
                   To     : in out Queue'Class;
                   Number : in     Natural := 1) is
   Person : Person_Name;
begin
   for I in 1..Number loop
      Remove_First(From, Person);
      Append(To, Person);
   end loop;
end Transfer;
```

This defines a Queue interface defining a queue of people. (A similar design could be created to define any kind of queue simply by replacing Person_Name by an appropriate type.) The Queue interface has four dispatching operations, Append, Remove_First, Cur_Count, and Max_Count. The body of a class-wide operation, Transfer is also shown. Every non-abstract extension of Queue must provide implementations for at least its four dispatching operations, as they are abstract. Any object of a type derived from Queue may be passed to Transfer as either the From or the To operand. The two operands need not be of the same type in any given call.

The Synchronized_Queue interface inherits the four dispatching operations from Queue and adds two additional dispatching operations, which wait if necessary rather than raising the Queue_Error exception. This synchronized interface may only be implemented by a task or protected type, and as such ensures safe concurrent access.

Example use of the interface:

```
type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue; Person : in Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;

...

Cashier, Counter : Fast_Food_Queue;

...
-- Add George (see 3.10.1) to the cashier's queue:
Append (Cashier, George);
-- After payment, move George to the sandwich counter queue:
Transfer (Cashier, Counter);
...
```

An interface such as Queue can be used directly as the parent of a new type (as shown here), or can be used as a progenitor when a type is derived. In either case, the primitive operations of the interface are inherited. For Queue, the implementation of the four inherited routines must be provided. Inside the call of Transfer, calls will dispatch to the implementations of Append and Remove_First for type Fast_Food_Queue.

Example of a task interface:

```
type Serial_Device is task interface;  -- see 9.1
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
procedure Write(Dev : in Serial_Device; C : in  Character) is abstract;
```

The Serial_Device interface has two dispatching operations which are intended to be implemented by task entries (see 9.1).

3.10 Access Types

Replace paragraph 2:

access_type_definition ::=
  access_to_object_definition
 | access_to_subprogram_definition

by:

access_type_definition ::=
  [null_exclusion] access_to_object_definition
 | [null_exclusion] access_to_subprogram_definition

Replace paragraph 6:

access_definition ::= **access** subtype_mark

by:

null_exclusion ::= **not null**

access_definition ::=
  [null_exclusion] **access** [**constant**] subtype_mark
 | [null_exclusion] **access** [**protected**] **procedure** parameter_profile
 | [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

Replace paragraph 9:

A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. Finally, the current instance of a limited type, and a formal parameter or generic formal object of a tagged type are defined to be aliased. Aliased views are the ones that can be designated by an access value. If the view defined by an object_declaration is aliased, and the type of the object has discriminants, then the object is constrained; if its nominal subtype is unconstrained, then the object is constrained by its initial value. Similarly, if the object created by an allocator has discriminants, the object is constrained, either by the designated subtype, or by its initial value.

by:

A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. The current instance of a limited tagged type, a protected type, a task type, or a type that has the reserved word **limited** in its full definition is also defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value.

Replace paragraph 12:

An access_definition defines an anonymous general access-to-variable type; the subtype_mark denotes its *designated subtype*. An access_definition is used in the specification of an access discriminant (see 3.7) or an access parameter (see 6.1).

by:

An access_definition defines an anonymous general access type or an anonymous access-to-subprogram type. For a general access type, the subtype_mark denotes its *designated subtype*; if the general_access_modifier **constant** appears, the type is an access-to-constant type; otherwise it is an access-to-variable type. For an access-to-subprogram type, the parameter_profile or parameter_and_result_profile denotes its *designated profile*.

Replace paragraph 13:

For each (named) access type, there is a literal **null** which has a null access value designating no entity at all. The null value of a named access type is the default initial value of the type. Other values of an access type are obtained by evaluating an attribute_reference for the Access or Unchecked_Access attribute of an aliased view of an object or non-

intrinsic subprogram, or, in the case of a named access-to-object type, an allocator, which returns an access value designating a newly created object (see 3.10.2).

by:

For each access type, there is a null access value designating no entity at all, which can be obtained by (implicitly) converting the literal **null** to the access type. The null value of an access type is the default initial value of the type. Non-null values of an access-to-object type are obtained by evaluating an allocator, which returns an access value designating a newly created object (see 3.10.2), or in the case of a general access-to-object type, evaluating an attribute_reference for the Access or Unchecked_Access attribute of an aliased view of an object. Non-null values of an access-to-subprogram type are obtained by evaluating an attribute_reference for the Access attribute of a non-intrinsic subprogram.

A null_exclusion in a construct specifies that the null value does not belong to the access subtype defined by the construct, that is, the access subtype *excludes null*. In addition, the anonymous access subtype defined by the access_definition for a controlling access parameter (see 3.9.2) excludes null. Finally, for a subtype_indication without a null_exclusion, the subtype denoted by the subtype_indication excludes null if and only if the subtype denoted by the subtype_mark in the subtype_indication excludes null.

Insert after paragraph 14:

All subtypes of an access-to-subprogram type are constrained. The first subtype of a type defined by an access_definition or an access_to_object_definition is unconstrained if the designated subtype is an unconstrained array or discriminated subtype; otherwise it is constrained.

the new paragraph:

Legality Rules

If a subtype_indication, discriminant_specification, parameter_specification, parameter_and_result_profile, object_renaming_declaration, or formal_object_declaration has a null_exclusion, the subtype_mark in that construct shall denote an access subtype that does not exclude null.

Replace paragraph 15:

A composite_constraint is *compatible* with an unconstrained access subtype if it is compatible with the designated subtype. An access value *satisfies* a composite_constraint of an access subtype if it equals the null value of its type or if it designates an object whose value satisfies the constraint.

by:

A composite_constraint is *compatible* with an unconstrained access subtype if it is compatible with the designated subtype. A null_exclusion is compatible with any access subtype that does not exclude null. An access value *satisfies* a composite_constraint of an access subtype if it equals the null value of its type or if it designates an object whose value satisfies the constraint. An access value satisfies an exclusion of the null value if it does not equal the null value of its type.

Replace paragraph 17:

The elaboration of an access_definition creates an anonymous general access-to-variable type [(this happens as part of the initialization of an access parameter or access discriminant)].

by:

The elaboration of an access_definition creates an anonymous access type.

Replace paragraph 22:

```
type Peripheral_Ref is access Peripheral;  -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
                                           -- general access-to-class-wide, see 3.9.1
```

by:

```
type Peripheral_Ref is not null access Peripheral;  -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
                                           -- general access-to-class-wide, see 3.9.1
```

3.10.1 Incomplete Type Declarations

Replace paragraph 2:

incomplete_type_declaration ::= **type** defining_identifier [discriminant_part];

by:

incomplete_type_declaration ::= **type** defining_identifier [discriminant_part] [**is tagged**];

Static Semantics

An incomplete_type_declaration declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a discriminant_part appears. If the incomplete_type_declaration includes the reserved word **tagged**, it declares a *tagged incomplete view*. An incomplete view of a type is a limited view of the type (see 7.5).

Given an access type *A* whose designated type *T* is an incomplete view, a dereference of a value of type *A* also has this incomplete view except when:

- it occurs within the immediate scope of the completion of *T*, or
- it occurs within the scope of a nonlimited_with_clause that mentions a library package in whose visible part the completion of *T* is declared.

In these cases, the dereference has the full view of *T*.

Similarly, if a subtype_mark denotes a subtype_declaration defining a subtype of an incomplete view *T*, the subtype_mark denotes an incomplete view except under the same two circumstances given above, in which case it denotes the full view of *T*.

Replace paragraph 4:

If an incomplete_type_declaration has a known_discriminant_part, then a full_type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). If an incomplete_type_declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding full_type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.

by:

If an incomplete_type_declaration includes the reserved word **tagged**, then a full_type_declaration that completes it shall declare a tagged type. If an incomplete_type_declaration has a known_discriminant_part, then a full_type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). If an incomplete_type_declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding full_type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.

Replace paragraph 5:

The only allowed uses of a name that denotes an incomplete_type_declaration are as follows:

by:

A name that denotes an incomplete view of a type may be used as follows:

Replace paragraph 7:

- as the subtype_mark defining the subtype of a parameter or result of an access_to_subprogram_definition;

by:

- as the subtype_mark in the subtype_indication of a subtype_declaration; the subtype_indication shall not have a null_exclusion or a constraint;

Replace paragraph 8:

- as the subtype_mark in an access_definition;

by:

- as the subtype_mark in an access_definition.

If such a name denotes a tagged incomplete view, it may also be used:

- as the subtype_mark defining the subtype of a parameter in a formal_part;

Replace paragraph 9:

- as the prefix of an attribute_reference whose attribute_designator is Class; such an attribute_reference is similarly restricted to the uses allowed here; when used in this way, the corresponding full_type_declaration shall declare a tagged type, and the attribute_reference shall occur in the same library unit as the incomplete_type_declaration.

by:

- as the prefix of an attribute_reference whose attribute_designator is Class; such an attribute_reference is restricted to the uses allowed here; it denotes a tagged incomplete view.

If such a name occurs within the declaration list containing the completion of the incomplete view, it may also be used:

- as the subtype_mark defining the subtype of a parameter or result of an access_to_subprogram_definition.

If any of the above uses occurs as part of the declaration of a primitive subprogram of the incomplete view, and the declaration occurs immediately within the private part of a package, then the completion of the incomplete view shall also occur immediately within the private part; it shall not be deferred to the package body.

No other uses of a name that denotes an incomplete view of a type are allowed.

Replace paragraph 10:

A dereference (whether implicit or explicit — see 4.1) shall not be of an incomplete type.

by:

A prefix that denotes an object shall not be of an incomplete view.

Delete paragraph 11:

An incomplete_type_declaration declares an incomplete type and its first subtype; the first subtype is unconstrained if a known_discriminant_part appears.

Replace paragraph 19:

```
type Person(<>);     -- incomplete type declaration
type Car;            -- incomplete type declaration
```

by:

```
type Person(<>);     -- incomplete type declaration
type Car is tagged; -- incomplete type declaration
```

Replace paragraph 20:

```
type Person_Name is access Person;
type Car_Name    is access all Car;
```

by:

```
type Person_Name is access Person;
type Car_Name    is access all Car'Class;
```

Replace paragraph 21:

```
type Car is
   record
      Number  : Integer;
      Owner   : Person_Name;
   end record;
```

by:

```
type Car is tagged
   record
```

```
      Number  : Integer;
      Owner   : Person_Name;
   end record;
```

3.10.2 Operations of Access Types

Replace paragraph 2:

For an attribute_reference with attribute_designator Access (or Unchecked_Access — see 13.10), the expected type shall be a single access type; the prefix of such an attribute_reference is never interpreted as an implicit_dereference. If the expected type is an access-to-subprogram type, then the expected profile of the prefix is the designated profile of the access type.

by:

For an attribute_reference with attribute_designator Access (or Unchecked_Access — see 13.10), the expected type shall be a single access type *A* such that:

- *A* is an access-to-object type with designated type *D* and the type of the prefix is *D*'Class or is covered by *D*, or
- *A* is an access-to-subprogram type whose designated profile is type conformant with that of the prefix.

The prefix of such an attribute_reference is never interpreted as an implicit_dereference or a parameterless function_call (see 4.1.4). The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix.

Replace paragraph 3:

The accessibility rules, which prevent dangling references, are written in terms of *accessibility levels*, which reflect the run-time nesting of *masters*. As explained in 7.6.1, a master is the execution of a task_body, a block_statement, a subprogram_body, an entry_body, or an accept_statement. An accessibility level is *deeper than* another if it is more deeply nested at run time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate an object that no longer exists. The Unchecked_Access attribute may be used to circumvent the accessibility rules.

by:

The accessibility rules, which prevent dangling references, are written in terms of *accessibility levels*, which reflect the run-time nesting of *masters*. As explained in 7.6.1, a master is the execution of a certain construct, such as a subprogram_body. An accessibility level is *deeper than* another if it is more deeply nested at run time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate an object that no longer exists. The Unchecked_Access attribute may be used to circumvent the accessibility rules.

Replace paragraph 7:

- An entity or view created by a declaration has the same accessibility level as the innermost enclosing master, except in the cases of renaming and derived access types described below. A parameter of a master has the same accessibility level as the master.

by:

- An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. A parameter of a master has the same accessibility level as the master.

Replace paragraph 9:

- The accessibility level of a view conversion is the same as that of the operand.

by:

- The accessibility level of a view conversion, qualified_expression, or parenthesized expression, is the same as that of the operand.

Replace paragraph 10:

- For a function whose result type is a return-by-reference type, the accessibility level of the result object is the same as that of the master that elaborated the function body. For any other function, the accessibility level of the result object is that of the execution of the called function.

by:

- The accessibility level of an aggregate or the result of a function call (or equivalent use of an operator) that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an aggregate or the result of a function call is that of the innermost master that evaluates the aggregate or function call.
- Within a return statement, the accessibility level of the return object is that of the execution of the return statement. If the return statement completes normally by returning from the function, then prior to leaving the function, the accessibility level of the return object changes to be a level determined by the point of call, as does the level of any coextensions (see below) of the return object.

Replace paragraph 12:

- The accessibility level of the anonymous access type of an access discriminant is the same as that of the containing object or associated constrained subtype.

by:

- The accessibility level of the anonymous access type defined by an access_definition of an object_renaming_declaration is the same as that of the renamed view.
- The accessibility level of the anonymous access type of an access discriminant in the subtype_indication or qualified_expression of an allocator, or in the expression or return_subtype_indication of a return statement is determined as follows:
  - If the value of the access discriminant is determined by a discriminant_association in a subtype_indication, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);
  - If the value of the access discriminant is determined by a component_association in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);
  - In other cases, where the value of the access discriminant is determined by an object with an unconstrained nominal subtype, the accessibility level of the object.
- The accessibility level of the anonymous access type of an access discriminant in any other context is that of the enclosing object.

Replace paragraph 13:

- The accessibility level of the anonymous access type of an access parameter is the same as that of the view designated by the actual. If the actual is an allocator, this is the accessibility level of the execution of the called subprogram.

by:

- The accessibility level of the anonymous access type of an access parameter specifying an access-to-object type is the same as that of the view designated by the actual.
- The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is deeper than that of any master; all such anonymous access types have this same level.

Replace paragraph 14:

- The accessibility level of an object created by an allocator is the same as that of the access type.

by:

- The accessibility level of an object created by an allocator is the same as that of the access type, except for an allocator of an anonymous access type that defines the value of an access parameter or an access discriminant. For an allocator defining the value of an access parameter, the accessibility level is that of the innermost master of the call. For one defining an access discriminant, the accessibility level is determined as follows:
  - for an allocator used to define the constraint in a subtype_declaration, the level of the subtype_declaration;
  - for an allocator used to define the constraint in a component_definition, the level of the enclosing type;
  - for an allocator used to define the discriminant of an object, the level of the object.

  In this last case, the allocated object is said to be a *coextension* of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. All coextensions of an object are finalized when the object is finalized (see 7.6.1).

Insert after paragraph 16:

- The accessibility level of a component, protected subprogram, or entry of (a view of) a composite object is the same as that of (the view of) the composite object.

the new paragraph:

In the above rules, the operand of a view conversion, parenthesized expression or qualified_expression is considered to be used in a context if the view conversion, parenthesized expression or qualified_expression itself is used in that context.

Replace paragraph 19:

- The statically deeper relationship does not apply to the accessibility level of the anonymous type of an access parameter; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

by:

- The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is statically deeper than that of any master; all such anonymous access types have this same level.
- The statically deeper relationship does not apply to the accessibility level of the anonymous type of an access parameter specifying an access-to-object type; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

Replace paragraph 26:

- The view shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased.

by:

- The view shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value.

Replace paragraph 27:

- If *A* is a named access type and *D* is a tagged type, then the type of the view shall be covered by *D*; if *A* is anonymous and *D* is tagged, then the type of the view shall be either *D*'Class or a type covered by D; if *D* is untagged, then the type of the view shall be *D*, and *A*'s designated subtype shall either statically match the nominal subtype of the view or be discriminated and unconstrained;

by:

- If *A* is a named access type and *D* is a tagged type, then the type of the view shall be covered by *D*; if *A* is anonymous and *D* is tagged, then the type of the view shall be either *D*'Class or a type covered by *D*; if *D* is untagged, then the type of the view shall be *D*, and either:
  - the designated subtype of *A* shall statically match the nominal subtype of the view; or
  - *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained.

Replace paragraph 32:

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of *S*. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. The profile of P shall be subtype-conformant with the designated profile of *S*, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic body, *S* shall be declared within the generic body.

by:

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of *S*. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. The profile of P shall be subtype-conformant with the designated profile of *S*, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic unit, and the expression P'Access occurs within the body of that generic unit or within the body of a generic unit declared within the declarative region of the generic unit, then the ultimate ancestor of *S* shall be either a non-formal type declared within the generic unit or an anonymous access type of an access parameter.

Replace paragraph 34:

82 The predefined operations of an access type also include the assignment operation, qualification, and membership tests. Explicit conversion is allowed between general access types with matching designated subtypes; explicit conversion is allowed between access-to-subprogram types with subtype conformant profiles (see 4.6). Named access types have predefined equality operators; anonymous access types do not (see 4.5.2).

by:

82 The predefined operations of an access type also include the assignment operation, qualification, and membership tests. Explicit conversion is allowed between general access types with matching designated subtypes; explicit conversion is allowed between access-to-subprogram types with subtype conformant profiles (see 4.6). Named access types have predefined equality operators; anonymous access types do not, but they can use the predefined equality operators for *universal_access* (see 4.5.2).

Replace paragraph 37:

The accessibility rules imply that it is not possible to use the Access attribute to implement "downward closures" — that is, to pass a more-nested subprogram as a parameter to a less-nested subprogram, as might be desired for example for an iterator abstraction. Instead, downward closures can be implemented using generic formal subprograms (see 12.6). Note that Unchecked_Access is not allowed for subprograms.

by:

88 The Access attribute for subprograms and parameters of an anonymous access-to-subprogram type may together be used to implement "downward closures" — that is, to pass a more-nested subprogram as a parameter to a less-nested subprogram, as might be appropriate for an iterator abstraction or numerical integration. Downward closures can also be implemented using generic formal subprograms (see 12.6). Note that Unchecked_Access is not allowed for subprograms.

3.11 Declarative Parts

Insert after paragraph 6:

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

the new paragraph:

Static Semantics

The list of declarative_items of a declarative_part is called the *declaration list* of the declarative_part.

Section 4: Names and Expressions

4.1 Names

Replace paragraph 11:

The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a direct_name or a character_literal.

by:

The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a direct_name or a character_literal.

4.1.3 Selected Components

Insert after paragraph 9:

- The prefix shall resolve to denote an object or value of some task or protected type (after any implicit dereference). The selector_name shall resolve to denote an entry_declaration or subprogram_declaration occurring (implicitly or explicitly) within the visible part of that type. The selected_component denotes the corresponding entry, entry family, or protected subprogram.

the new paragraph:

- A view of a subprogram whose first formal parameter is of a tagged type or is an access parameter whose designated type is tagged:

  The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type *T* or class-wide type *T*'Class. The selector_name shall resolve to denote a view of a subprogram declared immediately within the declarative region in which an ancestor of the type *T* is declared. The first formal parameter of the subprogram shall be of type *T*, or a class-wide type that covers *T*, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the selected_component. The selected_component denotes a view of this subprogram that omits the first formal parameter. This view is called a *prefixed view* of the subprogram, and the prefix of the selected_component (after any implicit dereference) is called the *prefix* of the prefixed view.

Insert after paragraph 13:

If the prefix does not denote a package, then it shall be a direct_name or an expanded name, and it shall resolve to denote a program unit (other than a package), the current instance of a type, a block_statement, a loop_statement, or an accept_statement (in the case of an accept_statement or entry_body, no family index is allowed); the expanded name shall occur within the declarative region of this construct. Further, if this construct is a callable construct and the prefix denotes more than one such enclosing callable construct, then the expanded name is ambiguous, independently of the selector_name.

the new paragraph:

Legality Rules

For a subprogram whose first parameter is an access parameter, the prefix of any prefixed view shall denote an aliased view of an object.

For a subprogram whose first parameter is of mode **in out** or **out**, or of an anonymous access-to-variable type, the prefix of any prefixed view shall denote a variable.

In paragraph 17 replace:

```
Control.Seize       -- an entry of a protected object       (see 9.4)
```

by:

```
Cashier.Append      -- a prefixed view of a procedure       (see 3.9.4)
```

```
Control.Seize        -- an entry of a protected object        (see 9.4)
```

4.1.4 Attributes

Replace paragraph 14:

5 In general, the name in a prefix of an attribute_reference (or a range_attribute_reference) has to be resolved without using any context. However, in the case of the Access attribute, the expected type for the prefix has to be a single access type, and if it is an access-to-subprogram type (see 3.10.2) then the resolution of the name can use the fact that the profile of the callable entity denoted by the prefix has to be type conformant with the designated profile of the access type.

by:

5 In general, the name in a prefix of an attribute_reference (or a range_attribute_reference) has to be resolved without using any context. However, in the case of the Access attribute, the expected type for the attribute_reference has to be a single access type, and the resolution of the name can use the fact that the type of the object or the profile of the callable entity denoted by the prefix has to match the designated type or be type conformant with the designated profile of the access type.

4.2 Literals

Delete paragraph 2:

The expected type for a literal **null** shall be a single access type.

Delete paragraph 7:

A literal null shall not be of an anonymous access type, since such types do not have a null value (see 3.10).

Replace paragraph 8:

An integer literal is of type *universal_integer*. A real literal is of type *universal_real*.

by:

An integer literal is of type *universal_integer*. A real literal is of type *universal_real*. The literal **null** is of type *universal_access*.

4.3 Aggregates

Replace paragraph 3:

The expected type for an aggregate shall be a single nonlimited array type, record type, or record extension.

by:

The expected type for an aggregate shall be a single array type, record type, or record extension.

4.3.1 Record Aggregates

Replace paragraph 4:

```
record_component_association ::=
  [ component_choice_list => ] expression
```

by:

```
record_component_association ::=
    [component_choice_list =>] expression
  |  component_choice_list => <>
```

Replace paragraph 8:

The expected type for a record_aggregate shall be a single nonlimited record type or record extension.

by:

The expected type for a record_aggregate shall be a single record type or record extension.

Replace paragraph 16:

Each record_component_association shall have at least one associated component, and each needed component shall be associated with exactly one record_component_association. If a record_component_association has two or more associated components, all of them shall be of the same type.

by:

Each record_component_association other than an **others** choice with a <> shall have at least one associated component, and each needed component shall be associated with exactly one record_component_association. If a record_component_association with an expression has two or more associated components, all of them shall be of the same type.

Insert after paragraph 17:

If the components of a variant_part are needed, then the value of a discriminant that governs the variant_part shall be given by a static expression.

the new paragraph:

A record_component_association for a discriminant without a default_expression shall have an expression rather than <>.

Insert before paragraph 20:

The expression of a record_component_association is evaluated (and converted) once for each associated component.

the new paragraph:

For a record_component_association with an expression, the expression defines the value for the associated component(s). For a record_component_association with <>, if the component_declaration has a default_expression, that default_expression defines the value for the associated component(s); otherwise, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

Replace paragraph 27:

Example of component association with several choices:

by:

Examples of component associations with several choices:

Insert after paragraph 29:

```
   -- The allocator is evaluated twice: Succ and Pred designate different cells
```

the new paragraphs:

```
(Value => 0, Succ|Pred => <>)                    -- see 3.10.1
   -- Succ and Pred will be set to null
```

4.3.2 Extension Aggregates

Replace paragraph 4:

The expected type for an extension_aggregate shall be a single nonlimited type that is a record extension. If the ancestor_part is an expression, it is expected to be of any nonlimited tagged type.

by:

The expected type for an extension_aggregate shall be a single type that is a record extension. If the ancestor_part is an expression, it is expected to be of any tagged type.

Replace paragraph 5:

If the ancestor_part is a subtype_mark, it shall denote a specific tagged subtype. The type of the extension_aggregate shall be derived from the type of the ancestor_part, through one or more record extensions (and no private extensions).

by:

If the ancestor_part is a subtype_mark, it shall denote a specific tagged subtype. If the ancestor_part is an expression, it shall not be dynamically tagged. The type of the extension_aggregate shall be derived from the type of the ancestor_part, through one or more record extensions (and no private extensions).

4.3.3 Array Aggregates

Replace paragraph 3:

```
positional_array_aggregate ::=
   (expression, expression {, expression})
  | (expression {, expression}, others => expression)
```

by:

```
positional_array_aggregate ::=
   (expression, expression {, expression})
  | (expression {, expression}, others => expression)
  | (expression {, expression}, others => <>)
```

Replace paragraph 5:

```
array_component_association ::=
   discrete_choice_list => expression
```

by:

```
array_component_association ::=
   discrete_choice_list => expression
  | discrete_choice_list => <>
```

Replace paragraph 7:

The expected type for an array_aggregate (that is not a subaggregate) shall be a single nonlimited array type. The component type of this array type is the expected type for each array component expression of the array_aggregate.

by:

The expected type for an array_aggregate (that is not a subaggregate) shall be a single array type. The component type of this array type is the expected type for each array component expression of the array_aggregate.

Replace paragraph 11:

For an explicit_actual_parameter, an explicit_generic_actual_parameter, the expression of a return_statement, the initialization expression in an object_declaration, or a default_expression (for a parameter or a component), when the nominal subtype of the corresponding formal parameter, generic formal parameter, function result, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

by:

For an explicit_actual_parameter, an explicit_generic_actual_parameter, the expression of a return statement, the initialization expression in an object_declaration, or a default_expression (for a parameter or a component), when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

Insert before paragraph 24:

The bounds of the index range of an array_aggregate (including a subaggregate) are determined as follows:

the new paragraph:

Each expression in an array_component_association defines the value for the associated component(s). For an array_component_association with <>, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

Insert after paragraph 43:

```
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"
```

the new paragraphs:

Example of an array aggregate with defaulted others choice and with an applicable index constraint provided by an enclosing record aggregate:

```
Buffer'(Size => 50, Pos => 1, Value => String'('x', others => <>))  -- see 3.7
```

4.4 Expressions

Replace paragraph 15:

```
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)
```

by:

```
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
R*Sin(θ)*Cos(φ)             -- 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)
```

4.5.2 Relational Operators and Membership Tests

Replace paragraph 3:

The *tested type* of a membership test is the type of the range or the type determined by the subtype_mark. If the tested type is tagged, then the simple_expression shall resolve to be of a type that covers or is covered by the tested type; if untagged, the expected type for the simple_expression is the tested type.

by:

The *tested type* of a membership test is the type of the range or the type determined by the subtype_mark. If the tested type is tagged, then the simple_expression shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type for the simple_expression is the tested type.

Insert after paragraph 7:

```
function "=" (Left, Right : T) return Boolean
function "/="(Left, Right : T) return Boolean
```

the new paragraphs:

The following additional equality operators for the *universal_access* type are declared in package Standard for use with anonymous access types:

```
function "=" (Left, Right : universal_access) return Boolean
function "/="(Left, Right : universal_access) return Boolean
```

Insert after paragraph 9:

```
function "<" (Left, Right : T) return Boolean
function "<="(Left, Right : T) return Boolean
function ">" (Left, Right : T) return Boolean
function ">="(Left, Right : T) return Boolean
```

the new paragraphs:

Name Resolution Rules

At least one of the operands of an equality operator for *universal_access* shall be of a specific anonymous access type. Unless the predefined equality operator is identified using an expanded name with prefix denoting the package Standard, neither operand shall be of an access-to-object type whose designated type is *D* or *D*'Class, where *D* has a user-defined primitive equality operator such that:

- its result type is Boolean;
- it is declared immediately within the same declaration list as *D*; and
- at least one of its operands is an access parameter with designated type *D*.

Legality Rules

At least one of the operands of the equality operators for *universal_access* shall be of type *universal_access*, or both shall be of access-to-object types, or both shall be of access-to-subprogram types. Further:

- When both are of access-to-object types, the designated types shall be the same or one shall cover the other, and if the designated types are elementary or array types, then the designated subtypes shall statically match;
- When both are of access-to-subprogram types, the designated profiles shall be subtype conformant.

Replace paragraph 30:

- The tested type is not scalar, and the value of the simple_expression satisfies any constraints of the named subtype, and, if the type of the simple_expression is class-wide, the value has a tag that identifies a type covered by the tested type.

by:

- The tested type is not scalar, and the value of the simple_expression satisfies any constraints of the named subtype, and:
  - if the type of the simple_expression is class-wide, the value has a tag that identifies a type covered by the tested type;
  - if the tested type is an access type and the named subtype excludes null, the value of the simple_expression is not null.

Delete paragraph 33:

13 No exception is ever raised by a membership test, by a predefined ordering operator, or by a predefined equality operator for an elementary type, but an exception can be raised by the evaluation of the operands. A predefined equality operator for a composite type can only raise an exception if the type has a tagged part whose primitive equals operator propagates an exception.

4.5.5 Multiplying Operators

Replace paragraph 20:

Legality Rules

The above two fixed-fixed multiplying operators shall not be used in a context where the expected type for the result is itself *universal_fixed* — the context has to identify some other numeric type to which the result is to be converted, either explicitly or implicitly.

by:

Name Resolution Rules

The above two fixed-fixed multiplying operators shall not be used in a context where the expected type for the result is itself *universal_fixed* — the context has to identify some other numeric type to which the result is to be converted, either explicitly or implicitly. Unless the predefined universal operator is identified using an expanded name with prefix denoting the package Standard, an explicit conversion is required on the result when using the above fixed-fixed multiplication operator if either operand is of a type having a user-defined primitive multiplication operator such that:

- it is declared immediately within the same declaration list as the type; and
- both of its formal parameters are of a fixed-point type.

A corresponding requirement applies to the universal fixed-fixed division operator.

4.6 Type Conversions

Replace paragraph 5:

A type_conversion whose operand is the name of an object is called a *view conversion* if both its target type and operand type are tagged, or if it appears as an actual parameter of mode **out** or **in out**; other type_conversions are called *value conversions*.

by:

A type_conversion whose operand is the name of an object is called a *view conversion* if both its target type and operand type are tagged, or if it appears in a call as an actual parameter of mode **out** or **in out**; other type_conversions are called *value conversions*.

Replace paragraph 8:

If the target type is a numeric type, then the operand type shall be a numeric type.

by:

In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type.

Delete paragraph 9:

If the target type is an array type, then the operand type shall be an array type. Further:

Delete paragraph 10:

- The types shall have the same dimensionality;

Delete paragraph 11:

- Corresponding index types shall be convertible;

Delete paragraph 12:

- The component subtypes shall statically match; and

Delete paragraph 12.1:

- In a view conversion, the target type and the operand type shall both or neither have aliased components.

Delete paragraph 13:

If the target type is a general access type, then the operand type shall be an access-to-object type. Further:

Delete paragraph 14:

- If the target type is an access-to-variable type, then the operand type shall be an access-to-variable type;

Delete paragraph 15:

- If the target designated type is tagged, then the operand designated type shall be convertible to the target designated type;

Delete paragraph 16:

- If the target designated type is not tagged, then the designated types shall be the same, and either the designated subtypes shall statically match or the target designated subtype shall be discriminated and unconstrained; and

Delete paragraph 17:

- The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Delete paragraph 18:

If the target type is an access-to-subprogram type, then the operand type shall be an access-to-subprogram type. Further:

Delete paragraph 19:

- The designated profiles shall be subtype-conformant.

Delete paragraph 20:

- The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

Replace paragraph 21:

If the target type is not included in any of the above four cases, there shall be a type that is an ancestor of both the target type and the operand type. Further, if the target type is tagged, then either:

by:

If there is a type that is an ancestor of both the target type and the operand type, or both types are class-wide types, then at least one of the following rules shall apply:

- The target type shall be untagged; or

Replace paragraph 23:

- The operand type shall be a class-wide type that covers the target type.

by:

- The operand type shall be a class-wide type that covers the target type; or

- The operand and target types shall both be class-wide types and the specific type associated with at least one of them shall be an interface type.

Replace paragraph 24:

In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type.

by:

If there is no type that is the ancestor of both the target type and the operand type, and they are not both class-wide types, one of the following rules shall apply:

- If the target type is a numeric type, then the operand type shall be a numeric type.
- If the target type is an array type, then the operand type shall be an array type. Further:
  - The types shall have the same dimensionality;
  - Corresponding index types shall be convertible;
  - The component subtypes shall statically match;
  - If the component types are anonymous access types, then the accessibility level of the operand type shall not be statically deeper than that of the target type;
  - Neither the target type nor the operand type shall be limited;
  - If the target type of a view conversion has aliased components, then so shall the operand type; and
  - The operand type of a view conversion shall not have a tagged, private, or volatile subcomponent.
- If the target type is *universal_access*, then the operand type shall be an access type.
- If the target type is a general access-to-object type, then the operand type shall be *universal_access* or an access-to-object type. Further, if the operand type is not *universal_access*:
  - If the target type is an access-to-variable type, then the operand type shall be an access-to-variable type;
  - If the target designated type is tagged, then the operand designated type shall be convertible to the target designated type;
  - If the target designated type is not tagged, then the designated types shall be the same, and either:
    - the designated subtypes shall statically match; or
    - the designated type shall be discriminated in its full view and unconstrained in any partial view, and one of the designated subtypes shall be unconstrained;
  - The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.
- If the target type is a pool-specific access-to-object type, then the operand type shall be *universal_access*.
- If the target type is an access-to-subprogram type, then the operand type shall be *universal_access* or an access-to-subprogram type. Further, if the operand type is not *universal_access*:
  - The designated profiles shall be subtype-conformant.
  - The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

Insert after paragraph 39:

- In either array case, the value of each component of the result is that of the matching component of the operand value (see 4.5.2).

the new paragraph:

- If the component types of the array types are anonymous access types, then a check is made that the accessibility level of the operand type is not deeper than that of the target type.

Replace paragraph 49:

- If the target type is an anonymous access type, a check is made that the value of the operand is not null; if the target is not an anonymous access type, then the result is null if the operand value is null.

by:

- If the operand value is null, the result of the conversion is the null value of the target type.

Replace paragraph 51:

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint.

by:

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null.

Replace paragraph 61:

22 A ramification of the overload resolution rules is that the operand of an (explicit) type_conversion cannot be the literal **null**, an allocator, an aggregate, a string_literal, a character_literal, or an attribute_reference for an Access or Unchecked_Access attribute. Similarly, such an expression enclosed by parentheses is not allowed. A qualified_expression (see 4.7) can be used instead of such a type_conversion.

by:

22 A ramification of the overload resolution rules is that the operand of an (explicit) type_conversion cannot be an allocator, an aggregate, a string_literal, a character_literal, or an attribute_reference for an Access or Unchecked_Access attribute. Similarly, such an expression enclosed by parentheses is not allowed. A qualified_expression (see 4.7) can be used instead of such a type_conversion.

4.8 Allocators

Replace paragraph 5:

If the type of the allocator is an access-to-constant type, the allocator shall be an initialized allocator. If the designated type is limited, the allocator shall be an uninitialized allocator.

by:

If the type of the allocator is an access-to-constant type, the allocator shall be an initialized allocator.

If the designated type of the type of the allocator is class-wide, the accessibility level of the type determined by the subtype_indication or qualified_expression shall not be statically deeper than that of the type of the allocator.

If the designated subtype of the type of the allocator has one or more unconstrained access discriminants, then the accessibility level of the anonymous access type of each access discriminant, as determined by the subtype_indication or qualified_expression of the allocator, shall not be statically deeper than that of the type of the allocator (see 3.10.2).

An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. This rule does not apply in the body of a generic unit or within a body declared within the declarative region of a generic unit, if the type of the allocator is a descendant of a formal access type declared within the formal part of the generic unit.

Replace paragraph 6:

If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the created object is always constrained; if the designated subtype is

constrained, then it provides the constraint of the created object; otherwise, the object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

by:

If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the designated subtype when the designated subtype is constrained or there is a partial view of the designated type that is constrained; otherwise, the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

Replace paragraph 7:

For the evaluation of an allocator, the elaboration of the subtype_indication or the evaluation of the qualified_expression is performed first. For the evaluation of an initialized allocator, an object of the designated type is created and the value of the qualified_expression is converted to the designated subtype and assigned to the object.

by:

For the evaluation of an initialized allocator, the evaluation of the qualified_expression is performed first. An object of the designated type is created and the value of the qualified_expression is converted to the designated subtype and assigned to the object.

Replace paragraph 8:

For the evaluation of an uninitialized allocator:

by:

For the evaluation of an uninitialized allocator, the elaboration of the subtype_indication is performed first. Then:

Replace paragraph 10:

- If the designated type is composite, an object of the designated type is created with tag, if any, determined by the subtype_mark of the subtype_indication; any per-object constraints on subcomponents are elaborated (see 3.8) and any implicit initial values for the subcomponents of the object are obtained as determined by the subtype_indication and assigned to the corresponding subcomponents. A check is made that the value of the object belongs to the designated subtype. Constraint_Error is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order.

by:

- If the designated type is composite, an object of the designated type is created with tag, if any, determined by the subtype_mark of the subtype_indication. This object is then initialized by default (see 3.3.1) using the subtype_indication to determine its nominal subtype. A check is made that the value of the object belongs to the designated subtype. Constraint_Error is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order.

For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the accessibility level of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, is not deeper than that of the type of the allocator. If the designated subtype of the allocator has one or more unconstrained access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. Program_Error is raised if either such check fails.

Replace paragraph 11:

If the created object contains any tasks, they are activated (see 9.2). Finally, an access value that designates the created object is returned.

by:

If the object to be created by an allocator has a controlled or protected part, and the finalization of the collection of the type of the allocator (see 7.6.1) has started, Program_Error is raised.

If the object to be created by an allocator contains any tasks, and the master of the type of the allocator is completed, and all of the dependent tasks of the master are terminated (see 9.3), then Program_Error is raised.

If the created object contains any tasks, they are activated (see 9.2). Finally, an access value that designates the created object is returned.

Bounded (Run-Time) Errors

It is a bounded error if the finalization of the collection of the type (see 7.6.1) of the allocator has started. If the error is detected, Program_Error is raised. Otherwise, the allocation proceeds normally.

4.9 Static Expressions and Static Subtypes

Replace paragraph 26:

A *static subtype* is either a *static scalar subtype* or a *static string subtype*. A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal scalar type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. A static string subtype is an unconstrained string subtype whose index subtype and component subtype are static (and whose type is not a descendant of a formal array type), or a constrained string subtype formed by imposing a compatible static constraint on a static string subtype. In any case, the subtype of a generic formal object of mode **in out**, and the result subtype of a generic formal function, are not static.

by:

A *static subtype* is either a *static scalar subtype* or a *static string subtype*. A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. A static string subtype is an unconstrained string subtype whose index subtype and component subtype are static, or a constrained string subtype formed by imposing a compatible static constraint on a static string subtype. In any case, the subtype of a generic formal object of mode **in out**, and the result subtype of a generic formal function, are not static.

Insert after paragraph 31:

- A discriminant constraint is static if each expression of the constraint is static, and the subtype of each discriminant is static.

the new paragraph:

In any case, the constraint of the first subtype of a scalar formal type is neither static nor null.

Replace paragraph 35:

- If the expression is not part of a larger static expression, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small.

by:

- If the expression is not part of a larger static expression and the expression is expected to be of a single specific type, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small.

Replace paragraph 36:

- If the expression is of type *universal_real* and its expected type is a decimal fixed point type, then its value shall be a multiple of the *small* of the decimal type.

by:

- If the expression is of type *universal_real* and its expected type is a decimal fixed point type, then its value shall be a multiple of the *small* of the decimal type. This restriction does not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance).

Replace paragraph 37:

The last two restrictions above do not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance).

by:

In addition to the places where Legality Rules normally apply (see 12.3), the above restrictions also apply in the private part of an instance of a generic unit.

Replace paragraph 38:

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal scalar type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, any rounding shall be performed away from zero. If the expected type is a descendant of a formal scalar type, no special rounding or truncating is required — normal accuracy rules apply (see Annex G).

by:

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, the rounding performed is implementation-defined. If the expected type is a descendant of a formal type, or if the static expression appears in the body of an instance of a generic unit and the corresponding expression is nonstatic in the corresponding generic body, then no special rounding or truncating is required — normal accuracy rules apply (see Annex G).

Implementation Advice

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the rounding should be the same as the default rounding for the target system.

4.9.1 Statically Matching Constraints and Subtypes

Replace paragraph 1:

A constraint *statically matches* another constraint if both are null constraints, both are static and have equal corresponding bounds or discriminant values, or both are nonstatic and result from the same elaboration of a constraint of a subtype_indication or the same evaluation of a range of a discrete_subtype_definition.

by:

A constraint *statically matches* another constraint if:

- both are null constraints;
- both are static and have equal corresponding bounds or discriminant values;
- both are nonstatic and result from the same elaboration of a constraint of a subtype_indication or the same evaluation of a range of a discrete_subtype_definition; or
- both are nonstatic and come from the same formal_type_declaration.

Replace paragraph 2:

A subtype *statically matches* another subtype of the same type if they have statically matching constraints. Two anonymous access subtypes statically match if their designated subtypes statically match.

by:

A subtype *statically matches* another subtype of the same type if they have statically matching constraints, and, for access subtypes, either both or neither exclude null. Two anonymous access-to-object subtypes statically match if their designated subtypes statically match, and either both or neither exclude null, and either both or neither are access-to-constant. Two anonymous access-to-subprogram subtypes statically match if their designated profiles are subtype conformant, and either both or neither exclude null.

Section 5: Statements

Replace paragraph 2:

This section describes the general rules applicable to all statements. Some statements are discussed in later sections: Procedure_call_statements and return_statements are described in 6, "Subprograms". Entry_call_statements, requeue_statements, delay_statements, accept_statements, select_statements, and abort_statements are described in 9, "Tasks and Synchronization". Raise_statements are described in 11, "Exceptions", and code_statements in 13. The remaining forms of statements are presented in this section.

by:

This section describes the general rules applicable to all statements. Some statements are discussed in later sections: Procedure_call_statements and return statements are described in 6, "Subprograms". Entry_call_statements, requeue_statements, delay_statements, accept_statements, select_statements, and abort_statements are described in 9, "Tasks and Synchronization". Raise_statements are described in 11, "Exceptions", and code_statements in 13. The remaining forms of statements are presented in this section.

5.1 Simple and Compound Statements - Sequences of Statements

Replace paragraph 4:

```
simple_statement ::= null_statement
  | assignment_statement | exit_statement
  | goto_statement | procedure_call_statement
  | return_statement | entry_call_statement
  | requeue_statement | delay_statement
  | abort_statement | raise_statement
  | code_statement
```

by:

```
simple_statement ::= null_statement
  | assignment_statement | exit_statement
  | goto_statement | procedure_call_statement
  | simple_return_statement | entry_call_statement
  | requeue_statement | delay_statement
  | abort_statement | raise_statement
  | code_statement
```

Replace paragraph 5:

```
compound_statement ::=
    if_statement | case_statement
  | loop_statement | block_statement
  | accept_statement | select_statement
```

by:

```
compound_statement ::=
    if_statement | case_statement
  | loop_statement | block_statement
  | extended_return_statement
  | accept_statement | select_statement
```

Replace paragraph 14:

A *transfer of control* is the run-time action of an exit_statement, return_statement, goto_statement, or requeue_statement, selection of a terminate_alternative, raising of an exception, or an abort, which causes the next action performed to be one other than what would normally be expected from the other rules of the language. As explained in 7.6.1, a transfer of control can cause the execution of constructs to be completed and then left, which may trigger finalization.

by:

A *transfer of control* is the run-time action of an exit_statement, return statement, goto_statement, or requeue_statement, selection of a terminate_alternative, raising of an exception, or an abort, which causes the next action performed to be one other than what would normally be expected from the other rules of the language. As explained in 7.6.1, a transfer of control can cause the execution of constructs to be completed and then left, which may trigger finalization.

5.2 Assignment Statements

Replace paragraph 4:

The *variable_*name of an assignment_statement is expected to be of any nonlimited type. The expected type for the expression is the type of the target.

by:

The *variable_*name of an assignment_statement is expected to be of any type. The expected type for the expression is the type of the target.

Replace paragraph 5:

The target denoted by the *variable_*name shall be a variable.

by:

The target denoted by the *variable_*name shall be a variable of a nonlimited type.

Delete paragraph 16:

3 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 are always constrained; however, subcomponents of such objects may be unconstrained.

Section 6: Subprograms

6.1 Subprogram Declarations

Replace paragraph 2:

subprogram_declaration ::= subprogram_specification ;

by:

subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification ;

Delete paragraph 3:

abstract_subprogram_declaration ::= subprogram_specification **is abstract**;

Replace paragraph 4:

subprogram_specification ::=
  procedure defining_program_unit_name parameter_profile
 | **function** defining_designator parameter_and_result_profile

by:

subprogram_specification ::=
  procedure_specification
 | function_specification

procedure_specification ::= **procedure** defining_program_unit_name parameter_profile

function_specification ::= **function** defining_designator parameter_and_result_profile

Replace paragraph 10:

The sequence of characters in an operator_symbol shall correspond to an operator belonging to one of the six classes of operators defined in clause 4.5 (spaces are not allowed and the case of letters is not significant).

by:

The sequence of characters in an operator_symbol shall form a reserved word, a delimiter, or compound delimiter that corresponds to an operator belonging to one of the six categories of operators defined in clause 4.5.

Replace paragraph 13:

parameter_and_result_profile ::= [formal_part] **return** subtype_mark

by:

parameter_and_result_profile ::=
  [formal_part] **return** [null_exclusion] subtype_mark
 | [formal_part] **return** access_definition

Replace paragraph 15:

parameter_specification ::=
  defining_identifier_list : mode subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

by:

parameter_specification ::=
  defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

Replace paragraph 20:

A subprogram_declaration or a generic_subprogram_declaration requires a completion: a body, a renaming_declaration (see 8.5), or a **pragma** Import (see B.1). A completion is not allowed for an abstract_subprogram_declaration.

by:

A subprogram_declaration or a generic_subprogram_declaration requires a completion: a body, a renaming_declaration (see 8.5), or a pragma Import (see B.1). A completion is not allowed for an abstract_subprogram_declaration (see 3.9.3) or a null_procedure_declaration (see 6.7).

Replace paragraph 23:

The nominal subtype of a formal parameter is the subtype denoted by the subtype_mark, or defined by the access_definition, in the parameter_specification.

by:

The nominal subtype of a formal parameter is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_specification. The nominal subtype of a function result is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_and_result_profile.

Replace paragraph 24:

An *access parameter* is a formal **in** parameter specified by an access_definition. An access parameter is of an anonymous general access-to-variable type (see 3.10). Access parameters allow dispatching calls to be controlled by access values.

by:

An *access parameter* is a formal **in** parameter specified by an access_definition. An *access result type* is a function result type specified by an access_definition. An access parameter or result type is of an anonymous access type (see 3.10). Access parameters of an access-to-object type allow dispatching calls to be controlled by access values. Access parameters of an access-to-subprogram type permit calls to subprograms passed as parameters irrespective of their accessibility level.

Replace paragraph 27:

- For any access parameters, the designated subtype of the parameter type.

by:

- For any access parameters of an access-to-object type, the designated subtype of the parameter type.
- For any access parameters of an access-to-subprogram type, the subtypes of the profile of the parameter type.

Replace paragraph 28:

- For any result, the result subtype.

by:

- For any non-access result, the nominal subtype of the function result.
- For any access result type of an access-to-object type, the designated subtype of the result type.
- For any access result type of an access-to-subprogram type, the subtypes of the profile of the result type.

Replace paragraph 30:

A subprogram declared by an abstract_subprogram_declaration is abstract; a subprogram declared by a subprogram_declaration is not. See 3.9.3, "Abstract Types and Subprograms".

by:

A subprogram declared by an abstract_subprogram_declaration is abstract; a subprogram declared by a subprogram_declaration is not. See 3.9.3, "Abstract Types and Subprograms". Similarly, a procedure defined by a

null_procedure_declaration is a null procedure; a procedure declared by a subprogram_declaration is not. See 6.7, "Null Procedures".

An overriding_indicator is used to indicate whether overriding is intended. See 8.3.1, "Overriding Indicators".

Replace paragraph 31:

The elaboration of a subprogram_declaration or an abstract_subprogram_declaration has no effect.

by:

The elaboration of a subprogram_declaration has no effect.

6.3 Subprogram Bodies

Replace paragraph 2:

```
subprogram_body ::=
  subprogram_specification is
    declarative_part
  begin
    handled_sequence_of_statements
  end [designator];
```

by:

```
subprogram_body ::=
  [overriding_indicator]
  subprogram_specification is
    declarative_part
  begin
    handled_sequence_of_statements
  end [designator];
```

6.3.1 Conformance Rules

Replace paragraph 10:

- a subprogram declared immediately within a protected_body.

by:

- a subprogram declared immediately within a protected_body;
- any prefixed view of a subprogram (see 4.1.3).

Insert after paragraph 13:

- The default calling convention is *entry* for an entry.

the new paragraph:

- The calling convention for an anonymous access-to-subprogram parameter or anonymous access-to-subprogram result is *protected* if the reserved word **protected** appears in its definition and otherwise is the convention of the subprogram that contains the parameter.

Replace paragraph 15:

Two profiles are *type conformant* if they have the same number of parameters, and both have a result if either does, and corresponding parameter and result types are the same, or, for access parameters, corresponding designated types are the same.

by:

Two profiles are *type conformant* if they have the same number of parameters, and both have a result if either does, and corresponding parameter and result types are the same, or, for access parameters or access results, corresponding designated types are the same, or corresponding designated profiles are type conformant.

Replace paragraph 16:

Two profiles are *mode conformant* if they are type-conformant, and corresponding parameters have identical modes, and, for access parameters, the designated subtypes statically match.

by:

Two profiles are *mode conformant* if they are type-conformant, and corresponding parameters have identical modes, and, for access parameters or access result types, the designated subtypes statically match, or the designated profiles are subtype conformant.

Insert after paragraph 24:

Two discrete_subtype_definitions are *fully conformant* if they are both subtype_indications or are both ranges, the subtype_marks (if any) denote the same subtype, and the corresponding simple_expressions of the ranges (if any) fully conform.

the new paragraph:

The *prefixed view profile* of a subprogram is the profile obtained by omitting the first parameter of that subprogram. There is no prefixed view profile for a parameterless subprogram. For the purposes of defining subtype and mode conformance, the convention of a prefixed view profile is considered to match that of either an entry or a protected operation.

6.3.2 Inline Expansion of Subprograms

Insert after paragraph 6:

For each call, an implementation is free to follow or to ignore the recommendation expressed by the pragma.

the new paragraph:

An implementation may allow a pragma Inline that has an argument which is a direct_name denoting a subprogram_body of the same declarative_part.

6.4 Subprogram Calls

Replace paragraph 8:

The name or prefix given in a procedure_call_statement shall resolve to denote a callable entity that is a procedure, or an entry renamed as (viewed as) a procedure. The name or prefix given in a function_call shall resolve to denote a callable entity that is a function. When there is an actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value.

by:

The name or prefix given in a procedure_call_statement shall resolve to denote a callable entity that is a procedure, or an entry renamed as (viewed as) a procedure. The name or prefix given in a function_call shall resolve to denote a callable entity that is a function. The name or prefix shall not resolve to denote an abstract subprogram unless it is also a dispatching subprogram. When there is an actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value.

Replace paragraph 10:

For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association is evaluated (see 6.4.1). If a default_expression is used, an implicit parameter_association is assumed for this rule. These evaluations are done in an arbitrary order. The subprogram_body is then executed. Finally, if the subprogram completes normally, then after it is left, any necessary assigning back of formal to actual parameters occurs (see 6.4.1).

by:

For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association is evaluated (see 6.4.1). If a default_expression is used, an implicit parameter_association is assumed for this rule. These evaluations are done in an arbitrary order. The subprogram_body is then executed, or a call on an entry or protected subprogram is performed (see 3.9.2). Finally, if the subprogram completes normally, then after it is left, any necessary assigning back of formal to actual parameters occurs (see 6.4.1).

If the name or prefix of a subprogram call denotes a prefixed view (see 4.1.3), the subprogram call is equivalent to a call on the underlying subprogram, with the first actual parameter being provided by the prefix of the prefixed view (or the Access attribute of this prefix if the first formal parameter is an access parameter), and the remaining actual parameters given by the actual_parameter_part, if any.

Replace paragraph 11:

The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return_statement.

by:

The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return statement.

Replace paragraph 12:

A function_call denotes a constant, as defined in 6.5; the nominal subtype of the constant is given by the result subtype of the function.

by:

A function_call denotes a constant, as defined in 6.5; the nominal subtype of the constant is given by the nominal subtype of the function result.

6.5 Return Statements

Replace paragraph 1:

A return_statement is used to complete the execution of the innermost enclosing subprogram_body, entry_body, or accept_statement.

by:

A simple_return_statement or extended_return_statement (collectively called a *return statement*) is used to complete the execution of the innermost enclosing subprogram_body, entry_body, or accept_statement.

Replace paragraph 2:

return_statement ::= **return** [expression];

by:

simple_return_statement ::= **return** [expression];

extended_return_statement ::=
return defining_identifier : [**aliased**] return_subtype_indication [:= expression] [**do**
handled_sequence_of_statements
end return];

return_subtype_indication ::= subtype_indication | access_definition

Replace paragraph 3:

The expression, if any, of a return_statement is called the *return expression*. The *result subtype* of a function is the subtype denoted by the subtype_mark after the reserved word **return** in the profile of the function. The expected type for a return expression is the result type of the corresponding function.

by:

The *result subtype* of a function is the subtype denoted by the subtype_mark, or defined by the access_definition, after the reserved word **return** in the profile of the function. The expected type for the expression, if any, of a simple_return_statement is the result type of the corresponding function. The expected type for the expression of an extended_return_statement is that of the return_subtype_indication.

Replace paragraph 4:

A return_statement shall be within a callable construct, and it *applies to* the innermost one. A return_statement shall not be within a body that is within the construct to which the return_statement applies.

by:

A return statement shall be within a callable construct, and it *applies to* the innermost callable construct or extended_return_statement that contains it. A return statement shall not be within a body that is within the construct to which the return statement applies.

Replace paragraph 5:

A function body shall contain at least one return_statement that applies to the function body, unless the function contains code_statements. A return_statement shall include a return expression if and only if it applies to a function body.

by:

A function body shall contain at least one return statement that applies to the function body, unless the function contains code_statements. A simple_return_statement shall include an expression if and only if it applies to a function body. An extended_return_statement shall apply to a function body.

For an extended_return_statement that applies to a function body:

- If the result subtype of the function is defined by a subtype_mark, the return_subtype_indication shall be a subtype_indication. The type of the subtype_indication shall be the result type of the function. If the result subtype of the function is constrained, then the subtype defined by the subtype_indication shall also be constrained and shall statically match this result subtype. If the result subtype of the function is unconstrained, then the subtype defined by the subtype_indication shall be a definite subtype, or there shall be an expression.
- If the result subtype of the function is defined by an access_definition, the return_subtype_indication shall be an access_definition. The subtype defined by the access_definition shall statically match the result subtype of the function. The accessibility level of this anonymous access subtype is that of the result subtype.

For any return statement that applies to a function body:

- If the result subtype of the function is limited, then the expression of the return statement (if any) shall be an aggregate, a function call (or equivalent use of an operator), or a qualified_expression or parenthesized expression whose operand is one of these.
- If the result subtype of the function is class-wide, the accessibility level of the type of the expression of the return statement shall not be statically deeper than that of the master that elaborated the function body. If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression of the simple_return_statement or the return_subtype_indication, shall not be statically deeper than that of the master that elaborated the function body.

Static Semantics

Within an extended_return_statement, the *return object* is declared with the given defining_identifier, with the nominal subtype defined by the return_subtype_indication.

Replace paragraph 6:

For the execution of a return_statement, the expression (if any) is first evaluated and converted to the result subtype.

by:

For the execution of an extended_return_statement, the subtype_indication or access_definition is elaborated. This creates the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which might raise Constraint_Error — see 4.6); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see 3.3.1). If the nominal subtype is indefinite, the return object is constrained by its initial value.

For the execution of a simple_return_statement, the expression (if any) is first evaluated, converted to the result subtype, and then is assigned to the anonymous *return object*.

If the return object has any parts that are tasks, the activation of those tasks does not occur until after the function returns (see 9.2).

Delete paragraph 7:

If the result type is class-wide, then the tag of the result is the tag of the value of the expression.

Replace paragraph 8:

If the result type is a specific tagged type:

by:

If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the expression.

Delete paragraph 9:

- If it is limited, then a check is made that the tag of the value of the return expression identifies the result type. Constraint_Error is raised if this check fails.

Delete paragraph 10:

- If it is nonlimited, then the tag of the result is that of the result type.

Delete paragraph 11:

A type is a *return-by-reference* type if it is a descendant of one of the following:

Delete paragraph 12:

- a tagged limited type;

Delete paragraph 13:

- a task or protected type;

Delete paragraph 14:

- a nonprivate type with the reserved word **limited** in its declaration;

Delete paragraph 15:

- a composite type with a subcomponent of a return-by-reference type;

Delete paragraph 16:

- a private type whose full type is a return-by-reference type.

Delete paragraph 17:

If the result type is a return-by-reference type, then a check is made that the return expression is one of the following:

Delete paragraph 18:

- a name that denotes an object view whose accessibility level is not deeper than that of the master that elaborated the function body; or

Delete paragraph 19:

- a parenthesized expression or qualified_expression whose operand is one of these kinds of expressions.

Replace paragraph 20:

The exception Program_Error is raised if this check fails.

by:

If the result subtype of a function has one or more unconstrained access discriminants, a check is made that the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or the return_subtype_indication of the function, is not deeper than that of the master that elaborated the function body. If this check fails, Program_Error is raised.

Delete paragraph 21:

For a function with a return-by-reference result type the result is returned by reference; that is, the function call denotes a constant view of the object associated with the value of the return expression. For any other function, the result is returned by copy; that is, the converted value is assigned into an anonymous constant created at the point of the return_statement, and the function call denotes that object.

Replace paragraph 22:

Finally, a transfer of control is performed which completes the execution of the callable construct to which the return_statement applies, and returns to the caller.

by:

For the execution of an extended_return_statement, the handled_sequence_of_statements is executed. Within this handled_sequence_of_statements, the execution of a simple_return_statement that applies to the extended_return_statement causes a transfer of control that completes the extended_return_statement. Upon completion of a return statement that applies to a callable construct, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller.

In the case of a function, the function_call denotes a constant view of the return object.

Implementation Permissions

If the result subtype of a function is unconstrained, and a call on the function is used to provide the initial value of an object with a constrained nominal subtype, Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body) if, while elaborating the return_subtype_indication or evaluating the expression of a return statement that applies to the function body, it is determined that the value of the result will violate the constraint of the subtype of this object.

Replace paragraph 24:

```
return;                          -- in a procedure body, entry_body, or accept_statement
return Key_Value(Last_Index);    -- in a function body
```

by:

```
return;                          -- in a procedure body, entry_body,
                                 --  accept_statement, or extended_return_statement

return Key_Value(Last_Index);    -- in a function body

return Node : Cell do            -- in a function body, see 3.10.1 for Cell
   Node.Value := Result;
   Node.Succ := Next_Node;
end return;
```

6.5.1 Pragma No_Return

Insert new clause:

A pragma No_Return indicates that a procedure cannot return normally; it may propagate an exception or loop forever.

Syntax

The form of a pragma No_Return, which is a representation pragma (see 13.1), is as follows:

pragma No_Return(*procedure*_local_name{, *procedure*_local_name});

Legality Rules

Each *procedure_*local_name shall denote one or more procedures or generic procedures; the denoted entities are *non-returning*. The *procedure_*local_name shall not denote a null procedure nor an instance of a generic unit.

A return statement shall not apply to a non-returning procedure or generic procedure.

A procedure shall be non-returning if it overrides a dispatching non-returning procedure. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

If a renaming-as-body completes a non-returning procedure declaration, then the renamed procedure shall be non-returning.

Static Semantics

If a generic procedure is non-returning, then so are its instances. If a procedure declared within a generic unit is non-returning, then so are the corresponding copies of that procedure in instances.

Dynamic Semantics

If the body of a non-returning procedure completes normally, Program_Error is raised at the point of the call.

Examples

```
procedure Fail(Msg : String);  -- raises Fatal_Error exception
pragma No_Return(Fail);
   -- Inform compiler and reader that procedure never returns normally
```

6.7 Null Procedures

Insert new clause:

A null_procedure_declaration provides a shorthand to declare a procedure with an empty body.

Syntax

null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification **is null**;

Static Semantics

A null_procedure_declaration declares a *null procedure*. A completion is not allowed for a null_procedure_declaration.

Dynamic Semantics

The execution of a null procedure is invoked by a subprogram call. For the execution of a subprogram call on a null procedure, the execution of the subprogram_body has no effect.

The elaboration of a null_procedure_declaration has no effect.

Examples

```
procedure Simplify(Expr : in out Expression) is null; -- see 3.9
-- By default, Simplify does nothing, but it may be overridden in extensions of Expression
```

Section 7: Packages

7.1 Package Specifications and Declarations

Replace paragraph 5:

A package_declaration or generic_package_declaration requires a completion (a body) if it contains any declarative_item that requires a completion, but whose completion is not in its package_specification.

by:

A package_declaration or generic_package_declaration requires a completion (a body) if it contains any basic_declarative_item that requires a completion, but whose completion is not in its package_specification.

Replace paragraph 6:

The first list of declarative_items of a package_specification of a package other than a generic formal package is called the *visible part* of the package. The optional list of declarative_items after the reserved word **private** (of any package_specification) is called the *private part* of the package. If the reserved word **private** does not appear, the package has an implicit empty private part.

by:

The first list of basic_declarative_items of a package_specification of a package other than a generic formal package is called the *visible part* of the package. The optional list of basic_declarative_items after the reserved word **private** (of any package_specification) is called the *private part* of the package. If the reserved word **private** does not appear, the package has an implicit empty private part. Each list of basic_declarative_items of a package_specification forms a *declaration list* of the package.

7.3 Private Types and Private Extensions

Replace paragraph 3:

```
private_extension_declaration ::=
  type defining_identifier [discriminant_part] is
    [abstract] new ancestor_subtype_indication with private;
```

by:

```
private_extension_declaration ::=
  type defining_identifier [discriminant_part] is
    [abstract] [limited | synchronized] new ancestor_subtype_indication
    [and interface_list] with private;
```

Replace paragraph 6:

A private type is limited if its declaration includes the reserved word **limited**; a private extension is limited if its ancestor type is limited. If the partial view is nonlimited, then the full view shall be nonlimited. If a tagged partial view is limited, then the full view shall be limited. On the other hand, if an untagged partial view is limited, the full view may be limited or nonlimited.

by:

A private type is limited if its declaration includes the reserved word **limited**; a private extension is limited if its ancestor type is a limited type that is not an interface type, or if the reserved word **limited** or **synchronized** appears in its definition. If the partial view is nonlimited, then the full view shall be nonlimited. If a tagged partial view is limited, then the full view shall be limited. On the other hand, if an untagged partial view is limited, the full view may be limited or nonlimited.

Insert after paragraph 7:

If the partial view is tagged, then the full view shall be tagged. On the other hand, if the partial view is untagged, then the full view may be tagged or untagged. In the case where the partial view is untagged and the full view is tagged, no

derivatives of the partial view are allowed within the immediate scope of the partial view; derivatives of the full view are allowed.

the new paragraphs:

If a full type has a partial view that is tagged, then:

- the partial view shall be a synchronized tagged type (see 3.9.4) if and only if the full type is a synchronized tagged type;
- the partial view shall be a descendant of an interface type (see 3.9.4) if and only if the full type is a descendant of the interface type.

Insert after paragraph 8:

The *ancestor subtype* of a private_extension_declaration is the subtype defined by the ancestor_subtype_indication; the ancestor type shall be a specific tagged type. The full view of a private extension shall be derived (directly or indirectly) from the ancestor type. In addition to the places where Legality Rules normally apply (see 12.3), the requirement that the ancestor be specific applies also in the private part of an instance of a generic unit.

the new paragraph:

If the reserved word **limited** appears in a private_extension_declaration, the ancestor type shall be a limited type. If the reserved word **synchronized** appears in a private_extension_declaration, the ancestor type shall be a limited interface.

Insert after paragraph 10:

If a private extension inherits known discriminants from the ancestor subtype, then the full view shall also inherit its discriminants from the ancestor subtype, and the parent subtype of the full view shall be constrained if and only if the ancestor subtype is constrained.

the new paragraph:

If the full_type_declaration for a private extension is a derived_type_declaration, then the reserved word **limited** shall appear in the full_type_declaration if and only if it also appears in the private_extension_declaration.

Replace paragraph 16:

A private extension inherits components (including discriminants unless there is a new discriminant_part specified) and user-defined primitive subprograms from its ancestor type, in the same way that a record extension inherits components and user-defined primitive subprograms from its parent type (see 3.4).

by:

A private extension inherits components (including discriminants unless there is a new discriminant_part specified) and user-defined primitive subprograms from its ancestor type and its progenitor types (if any), in the same way that a record extension inherits components and user-defined primitive subprograms from its parent type and its progenitor types (see 3.4).

Replace paragraph 19:

Declaring a private type with an unknown_discriminant_part is a way of preventing clients from creating uninitialized objects of the type; they are then forced to initialize each object by calling some operation declared in the visible part of the package. If such a type is also limited, then no objects of the type can be declared outside the scope of the full_type_declaration, restricting all object creation to the package defining the type. This allows complete control over all storage allocation for the type. Objects of such a type can still be passed as parameters, however.

by:

Declaring a private type with an unknown_discriminant_part is a way of preventing clients from creating uninitialized objects of the type; they are then forced to initialize each object by calling some operation declared in the visible part of the package.

Replace paragraph 20:

7 The ancestor type specified in a private_extension_declaration and the parent type specified in the corresponding declaration of a record extension given in the private part need not be the same — the parent type of the full view can be any

descendant of the ancestor type. In this case, for a primitive subprogram that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions (if any) come from the corresponding primitive subprogram of the specified ancestor type, while the body comes from the corresponding primitive subprogram of the parent type of the full view. See 3.9.2.

by:

7 The ancestor type specified in a private_extension_declaration and the parent type specified in the corresponding declaration of a record extension given in the private part need not be the same. If the ancestor type is not an interface type, the parent type of the full view can be any descendant of the ancestor type. In this case, for a primitive subprogram that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions (if any) come from the corresponding primitive subprogram of the specified ancestor type, while the body comes from the corresponding primitive subprogram of the parent type of the full view. See 3.9.2.

8 If the ancestor type specified in a private_extension_declaration is an interface type, the parent type can be any type so long as the full view is a descendant of the ancestor type. The progenitor types specified in a private_extension_declaration and the progenitor types specified in the corresponding declaration of a record extension given in the private part need not be the same — the only requirement is that the private extension and the record extension be descended from the same set of interfaces.

7.3.1 Private Operations

Replace paragraph 12:

9 Partial views provide assignment (unless the view is limited), membership tests, selected components for the selection of discriminants and inherited components, qualification, and explicit conversion.

by:

9 Partial views provide initialization, membership tests, selected components for the selection of discriminants and inherited components, qualification, and explicit conversion. Nonlimited partial views also allow use of assignment_statements.

7.4 Deferred Constants

Replace paragraph 5:

- The deferred and full constants shall have the same type;

by:

- The deferred and full constants shall have the same type, or shall have statically matching anonymous access subtypes;

Replace paragraph 6:

- If the subtype defined by the subtype_indication in the deferred declaration is constrained, then the subtype defined by the subtype_indication in the full declaration shall match it statically. On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;

by:

- If the deferred constant declaration includes a subtype_indication that defines a constrained subtype, then the subtype defined by the subtype_indication in the full declaration shall match it statically. On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;

Replace paragraph 7:

- If the deferred constant declaration includes the reserved word **aliased**, then the full declaration shall also.

by:

- If the deferred constant declaration includes the reserved word **aliased**, then the full declaration shall also;

- If the subtype of the deferred constant declaration excludes null, the subtype of the full declaration shall also exclude null.

Replace paragraph 9:

The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4).

by:

The completion of a deferred constant declaration shall occur before the constant is frozen (see 13.14).

7.5 Limited Types

Replace paragraph 1:

A limited type is (a view of) a type for which the assignment operation is not allowed. A nonlimited type is a (view of a) type for which the assignment operation is allowed.

by:

A limited type is (a view of) a type for which copying (such as for an assignment_statement) is not allowed. A nonlimited type is a (view of a) type for which copying is allowed.

Replace paragraph 2:

If a tagged record type has any limited components, then the reserved word **limited** shall appear in its record_type_definition.

by:

If a tagged record type has any limited components, then the reserved word **limited** shall appear in its record_type_definition. If the reserved word **limited** appears in the definition of a derived_type_definition, its parent type and any progenitor interfaces shall be limited.

In the following contexts, an expression of a limited type is not permitted unless it is an aggregate, a function_call, or a parenthesized expression or qualified_expression whose operand is permitted by this rule:

- the initialization expression of an object_declaration (see 3.3.1)
- the default_expression of a component_declaration (see 3.8)
- the expression of a record_component_association (see 4.3.1)
- the expression for an ancestor_part of an extension_aggregate (see 4.3.2)
- an expression of a positional_array_aggregate or the expression of an array_component_association (see 4.3.3)
- the qualified_expression of an initialized allocator (see 4.8)
- the expression of a return statement (see 6.5)
- the default_expression or actual parameter for a formal object of mode **in** (see 12.4)

Replace paragraph 3:

A type is *limited* if it is a descendant of one of the following:

by:

A type is *limited* if it is one of the following:

Replace paragraph 4:

- a type with the reserved word **limited** in its definition;

by:

- a type with the reserved word **limited**, **synchronized**, **task**, or **protected** in its definition;

Delete paragraph 5:

- a task or protected type;

Replace paragraph 6:

- a composite type with a limited component.

the new paragraph:

- a composite type with a limited component;
- a derived type whose parent is limited and is not an interface.

Insert after paragraph 8:

There are no predefined equality operators for a limited type.

the new paragraph:

Implementation Requirements

For an aggregate of a limited type used to initialize an object as allowed above, the implementation shall not create a separate anonymous object for the aggregate. For a function_call of a type with a part that is of a task, protected, or explicitly limited record type that is used to initialize an object as allowed above, the implementation shall not create a separate return object (see 6.5) for the function_call. The aggregate or function_call shall be constructed directly in the new object.

Replace paragraph 9:

13 The following are consequences of the rules for limited types:

by:

13 While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation must be an aggregate or function_call, and such aggregates and function_calls must be built directly in the target object.

Delete paragraph 10:

- An initialization expression is not allowed in an object_declaration if the type of the object is limited.

Delete paragraph 11:

- A default expression is not allowed in a component_declaration if the type of the record component is limited.

Delete paragraph 12:

- An initialized allocator is not allowed if the designated type is limited.

Delete paragraph 13:

- A generic formal parameter of mode **in** must not be of a limited type.

Delete paragraph 14:

14 Aggregates are not available for a limited composite type. Concatenation is not available for a limited array type.

Delete paragraph 15:

15 The 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 declaration of the constant is of a nonlimited type.

Replace paragraph 23:

The fact that the full view of File_Name is explicitly declared **limited** means that parameter passing and function return will always be by reference (see 6.2 and 6.5).

by:

The fact that the full view of File_Name is explicitly declared **limited** means that parameter passing will always be by reference and function results will always be built directly in the result object (see 6.2 and 6.5).

7.6 User-Defined Assignment and Finalization

Replace paragraph 5:

```
type Controlled is abstract tagged private;
```

by:

```
type Controlled is abstract tagged private;
pragma Preelaborable_Initialization(Controlled);
```

Replace paragraph 6:

```
procedure Initialize (Object : in out Controlled);
procedure Adjust   (Object : in out Controlled);
procedure Finalize (Object : in out Controlled);
```

by:

```
procedure Initialize (Object : in out Controlled) is null;
procedure Adjust     (Object : in out Controlled) is null;
procedure Finalize   (Object : in out Controlled) is null;
```

Replace paragraph 7:

```
type Limited_Controlled is abstract tagged limited private;
```

by:

```
type Limited_Controlled is abstract tagged limited private;
pragma Preelaborable_Initialization(Limited_Controlled);
```

Replace paragraph 8:

```
    procedure Initialize (Object : in out Limited_Controlled);
    procedure Finalize (Object : in out Limited_Controlled);
private
    ... -- not specified by the language
end Ada.Finalization;
```

by:

```
    procedure Initialize (Object : in out Limited_Controlled) is null;
    procedure Finalize   (Object : in out Limited_Controlled) is null;
private
    ... -- not specified by the language
end Ada.Finalization;
```

Replace paragraph 9:

A controlled type is a descendant of Controlled or Limited_Controlled. The (default) implementations of Initialize, Adjust, and Finalize have no effect. The predefined "=" operator of type Controlled always returns True, since this operator is incorporated into the implementation of the predefined equality operator of types derived from Controlled, as explained in 4.5.2. The type Limited_Controlled is like Controlled, except that it is limited and it lacks the primitive subprogram Adjust.

by:

A controlled type is a descendant of Controlled or Limited_Controlled. The predefined "=" operator of type Controlled always returns True, since this operator is incorporated into the implementation of the predefined equality operator of types derived from Controlled, as explained in 4.5.2. The type Limited_Controlled is like Controlled, except that it is limited and it lacks the primitive subprogram Adjust.

A type is said to *need finalization* if:

- it is a controlled type, a task type or a protected type; or
- it has a component that needs finalization; or
- it is a limited type that has an access discriminant whose designated type needs finalization; or
- it is one of a number of language-defined types that are explicitly defined to need finalization.

Replace paragraph 10:

During the elaboration of an object_declaration, for every controlled subcomponent of the object that is not assigned an initial value (as defined in 3.3.1), Initialize is called on that subcomponent. Similarly, if the object as a whole is controlled and is not assigned an initial value, Initialize is called on the object. The same applies to the evaluation of an allocator, as explained in 4.8.

by:

During the elaboration or evaluation of a construct that causes an object to be initialized by default, for every controlled subcomponent of the object that is not assigned an initial value (as defined in 3.3.1), Initialize is called on that subcomponent. Similarly, if the object that is initialized by default as a whole is controlled, Initialize is called on the object.

Replace paragraph 11:

For an extension_aggregate whose ancestor_part is a subtype_mark, for each controlled subcomponent of the ancestor part, either Initialize is called, or its initial value is assigned, as appropriate Initialize is called on all controlled subcomponents of the ancestor part; if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

by:

For an extension_aggregate whose ancestor_part is a subtype_mark denoting a controlled subtype, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

Replace paragraph 17.1:

For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement or a return_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

by:

For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

Replace paragraph 21:

- For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an assignment_statement, the implementation need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object). Even if an anonymous object is created, the implementation may move its value to the target object as part of the assignment without re-adjusting so long as the anonymous object has no aliased subcomponents.

by:

- For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an assignment_statement, the implementation need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose

storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

Furthermore, an implementation is permitted to omit implicit Initialize, Adjust, and Finalize calls and associated assignment operations on an object of a nonlimited controlled type provided that:

- any omitted Initialize call is not a call on a user-defined Initialize procedure, and
- any usage of the value of the object after the implicit Initialize or Adjust call and before any subsequent Finalize call on the object does not change the external effect of the program, and
- after the omission of such calls and operations, any execution of the program that executes an Initialize or Adjust call on an object or initializes an object by an aggregate will also later execute a Finalize call on the object and will always do so prior to assigning a new value to the object, and
- the assignment operations associated with omitted Adjust calls are also omitted.

This permission applies to Adjust and Finalize calls even if the implicit calls have additional external effects.

7.6.1 Completion and Finalization

Replace paragraph 2:

The execution of a construct or entity is *complete* when the end of that execution has been reached, or when a transfer of control (see 5.1) causes it to be abandoned. Completion due to reaching the end of execution, or due to the transfer of control of an exit_, return_, goto_, or requeue_statement or of the selection of a terminate_alternative is *normal completion*. Completion is *abnormal* otherwise — when control is transferred out of a construct due to abort or the raising of an exception.

by:

The execution of a construct or entity is *complete* when the end of that execution has been reached, or when a transfer of control (see 5.1) causes it to be abandoned. Completion due to reaching the end of execution, or due to the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement or of the selection of a terminate_alternative is *normal completion*. Completion is *abnormal* otherwise — when control is transferred out of a construct due to abort or the raising of an exception.

Replace paragraph 3:

After execution of a construct or entity is complete, it is *left*, meaning that execution continues with the next action, as defined for the execution that is taking place. Leaving an execution happens immediately after its completion, except in the case of a *master*: the execution of a task_body, a block_statement, a subprogram_body, an entry_body, or an accept_statement. A master is finalized after it is complete, and before it is left.

by:

After execution of a construct or entity is complete, it is *left*, meaning that execution continues with the next action, as defined for the execution that is taking place. Leaving an execution happens immediately after its completion, except in the case of a *master*: the execution of a body other than a package_body; the execution of a statement; or the evaluation of an expression, function_call, or range that is not part of an enclosing expression, function_call, range, or simple_statement other than a simple_return_statement. A master is finalized after it is complete, and before it is left.

Replace paragraph 9:

- If the object is of a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations.

by:

- If the object is of a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations;
- If the object has coextensions (see 3.10.2), each coextension is finalized after the object whose access discriminant designates it.

Replace paragraph 11:

The order in which the finalization of a master performs finalization of objects is as follows: Objects created by declarations in the master are finalized in the reverse order of their creation. For objects that were created by allocators for an access type whose ultimate ancestor is declared in the master, this rule is applied as though each such object that still exists had been created in an arbitrary order at the first freezing point (see 13.14) of the ultimate ancestor type.

by:

The order in which the finalization of a master performs finalization of objects is as follows: Objects created by declarations in the master are finalized in the reverse order of their creation. For objects that were created by allocators for an access type whose ultimate ancestor is declared in the master, this rule is applied as though each such object that still exists had been created in an arbitrary order at the first freezing point (see 13.14) of the ultimate ancestor type; the finalization of these objects is called the *finalization of the collection*. After the finalization of a master is complete, the objects finalized as part of its finalization cease to *exist*, as do any types and subtypes defined and created within the master.

Replace paragraph 12:

The target of an assignment statement is finalized before copying in the new value, as explained in 7.6.

by:

The target of an assignment_statement is finalized before copying in the new value, as explained in 7.6.

Replace paragraph 13:

If the object_name in an object_renaming_declaration, or the actual parameter for a generic formal **in out** parameter in a generic_instantiation, denotes any part of an anonymous object created by a function call, the anonymous object is not finalized until after it is no longer accessible via any name. Otherwise, an anonymous object created by a function call or by an aggregate is finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, the object is finalized before starting the execution of any statement within the compound_statement.

by:

The master of an object is the master enclosing its creation whose accessibility level (see 3.10.2) is equal to that of the object.

Replace paragraph 13.1:

If a transfer of control or raising of an exception occurs prior to performing a finalization of an anonymous object, the anonymous object is finalized as part of the finalizations due to be performed for the object's innermost enclosing master.

by:

In the case of an expression that is a master, finalization of any (anonymous) objects occurs as the final part of evaluation of the expression.

Replace paragraph 16:

- For an Adjust invoked as part of the initialization of a controlled object, other adjustments due to be performed might or might not be performed, and then Program_Error is raised. During its propagation, finalization might or might not be applied to objects whose Adjust failed. For an Adjust invoked as part of an assignment statement, any other adjustments due to be performed are performed, and then Program_Error is raised.

by:

- For an Adjust invoked as part of assignment operations other than those invoked as part of an assignment_statement, other adjustments due to be performed might or might not be performed, and then Program_Error is raised. During its propagation, finalization might or might not be applied to objects whose Adjust failed. For an Adjust invoked as part of an assignment_statement, any other adjustments due to be performed are performed, and then Program_Error is raised.

Replace paragraph 18:

For a Finalize invoked by the transfer of control of an exit_, return_, goto_, or requeue_statement, Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Any other finalizations due to be performed up to that point are performed before raising Program_Error.

by:

For a Finalize invoked by the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement, Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Any other finalizations due to be performed up to that point are performed before raising Program_Error.

Section 8: Visibility Rules

8.1 Declarative Region

Insert after paragraph 4:

- a loop_statement;

the new paragraph:

- an extended_return_statement;

8.2 Scope of Declarations

Insert after paragraph 10:

The scope of a declaration always contains the immediate scope of the declaration. In addition, for a given declaration that occurs immediately within the visible part of an outer declaration, or is a public child of an outer declaration, the scope of the given declaration extends to the end of the scope of the outer declaration, except that the scope of a library_item includes only its semantic dependents.

the new paragraph:

The scope of an attribute_definition_clause is identical to the scope of a declaration that would occur at the point of the attribute_definition_clause.

8.3 Visibility

Insert after paragraph 12:

- An implicit declaration of an inherited subprogram overrides a previous implicit declaration of an inherited subprogram.

the new paragraphs:

- If two or more homographs are implicitly declared at the same place:
  - If at least one is a subprogram that is neither a null procedure nor an abstract subprogram, and does not require overriding (see 3.9.3), then they override those that are null procedures, abstract subprograms, or require overriding. If more than one such homograph remains that is not thus overridden, then they are all hidden from all visibility.
  - Otherwise (all are null procedures, abstract subprograms, or require overriding), then any null procedure overrides all abstract subprograms and all subprograms that require overriding; if more than one such homograph remains that is not thus overridden, then if they are all fully conformant with one another, one is chosen arbitrarily; if not, they are all hidden from all visibility.

Replace paragraph 18:

- For a package_declaration, task declaration, protected declaration, generic_package_declaration, or subprogram_body, the declaration is hidden from all visibility only until the reserved word **is** of the declaration.

by:

- For a package_declaration, generic_package_declaration, or subprogram_body, the declaration is hidden from all visibility only until the reserved word **is** of the declaration;
- For a task declaration or protected declaration, the declaration is hidden from all visibility only until the reserved word **with** of the declaration if there is one, or the reserved word **is** of the declaration if there is no **with**.

Replace paragraph 20:

- The declaration of a library unit (including a library_unit_renaming_declaration) is hidden from all visibility except at places that are within its declarative region or within the scope of a with_clause that mentions it. For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent. Such a nested declaration is hidden from all visibility except at places that are within the scope of a with_clause that mentions the child.

by:

- The declaration of a library unit (including a library_unit_renaming_declaration) is hidden from all visibility at places outside its declarative region that are not within the scope of a nonlimited_with_clause that mentions it. The limited view of a library package is hidden from all visibility at places that are not within the scope of a limited_with_clause that mentions it; in addition, the limited view is hidden from all visibility within the declarative region of the package, as well as within the scope of any nonlimited_with_clause that mentions the package. Where the declaration of the limited view of a package is visible, any name that denotes the package denotes the limited view, including those provided by a package renaming.
- For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent. Such a nested declaration is hidden from all visibility except at places that are within the scope of a with_clause that mentions the child.

Insert after paragraph 23:

- A declaration is also hidden from direct visibility where hidden from all visibility.

the new paragraph:

An attribute_definition_clause is *visible* everywhere within its scope.

Replace paragraph 26:

A non-overridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the non-overridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the context_clause for a subunit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

by:

A non-overridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the non-overridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the context_clause for a compilation unit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the compilation unit, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

8.3.1 Overriding Indicators

Insert new clause:

An overriding_indicator is used to declare that an operation is intended to override (or not override) an inherited operation.

Syntax

overriding_indicator ::= **[not] overriding**

Legality Rules

If an abstract_subprogram_declaration, null_procedure_declaration, subprogram_body, subprogram_body_stub, subprogram_renaming_declaration, generic_instantiation of a subprogram, or subprogram_declaration other than a protected subprogram has an overriding_indicator, then:

- the operation shall be a primitive operation for some type;
- if the overriding_indicator is **overriding**, then the operation shall override a homograph at the place of the declaration or body;
- if the overriding_indicator is **not overriding**, then the operation shall not override any homograph (at any place).

In addition to the places where Legality Rules normally apply, these rules also apply in the private part of an instance of a generic unit.

NOTES

8 Rules for overriding_indicators of task and protected entries and of protected subprograms are found in 9.5.2 and 9.4, respectively.

Examples

The use of overriding_indicators allows the detection of errors at compile-time that otherwise might not be detected at all. For instance, we might declare a security queue derived from the Queue interface of 3.9.4 as:

```
type Security_Queue is new Queue with record ...;

overriding
procedure Append(Q : in out Security_Queue; Person : in Person_Name);

overriding
procedure Remove_First(Q : in out Security_Queue; Person : in Person_Name);

overriding
function Cur_Count(Q : in Security_Queue) return Natural;

overriding
function Max_Count(Q : in Security_Queue) return Natural;

not overriding
procedure Arrest(Q : in out Security_Queue; Person : in Person_Name);
```

The first four subprogram declarations guarantee that these subprograms will override the four subprograms inherited from the Queue interface. A misspelling in one of these subprograms will be detected by the implementation. Conversely, the declaration of Arrest guarantees that this is a new operation.

8.4 Use Clauses

Replace paragraph 5:

A *package_*name of a use_package_clause shall denote a package.

by:

A *package_*name of a use_package_clause shall denote a nonlimited view of a package.

Insert after paragraph 7:

For a use_clause immediately within a declarative region, the scope is the portion of the declarative region starting just after the use_clause and extending to the end of the declarative region. However, the scope of a use_clause in the private part of a library unit does not include the visible part of any public descendant of that library unit.

the new paragraph:

A package is *named* in a use_package_clause if it is denoted by a *package_*name of that clause. A type is *named* in a use_type_clause if it is determined by a subtype_mark of that clause.

Replace paragraph 8:

For each package denoted by a *package*_name of a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class determined by a subtype_mark of a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place.

by:

For each package named in a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class named in a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place.

8.5.1 Object Renaming Declarations

Replace paragraph 2:

```
object_renaming_declaration ::=
   defining_identifier : subtype_mark renames object_name;
```

by:

```
object_renaming_declaration ::=
   defining_identifier : [null_exclusion] subtype_mark renames object_name;
 | defining_identifier : access_definition renames object_name;
```

Replace paragraph 3:

The type of the *object*_name shall resolve to the type determined by the subtype_mark.

by:

The type of the *object*_name shall resolve to the type determined by the subtype_mark, or in the case where the type is defined by an access_definition, to an anonymous access type. If the anonymous access type is an access-to-object type, the type of the *object*_name shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the *object*_name shall have a designated profile that is type conformant with that of the access_definition.

Insert after paragraph 4:

The renamed entity shall be an object.

the new paragraphs:

In the case where the type is defined by an access_definition, the type of the renamed object and the type defined by the access_definition:

- shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or
- shall both be access-to-subprogram types with subtype conformant designated profiles.

For an object_renaming_declaration with a null_exclusion or an access_definition that has a null_exclusion:

- if the *object*_name denotes a generic formal object of a generic unit *G*, and the object_renaming_declaration occurs within the body of *G* or within the body of a generic unit declared within the declarative region of *G*, then the declaration of the formal object of *G* shall have a null_exclusion;
- otherwise, the subtype of the *object*_name shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Replace paragraph 5:

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. A slice of an array shall not be renamed if this

restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. These rules also apply for a renaming that appears in the body of a generic unit, with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of an untagged generic formal derived type.

by:

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. These rules also apply for a renaming that appears in the body of a generic unit, with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of an untagged generic formal derived type.

Replace paragraph 6:

An object_renaming_declaration declares a new view of the renamed object whose properties are identical to those of the renamed view. Thus, 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 subtype_mark of the object_renaming_declaration is ignored).

by:

An object_renaming_declaration declares a new view of the renamed object whose properties are identical to those of the renamed view. Thus, 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 null exclusion or constraints that apply to an object are not affected by renaming (any constraint implied by the subtype_mark or access_definition of the object_renaming_declaration is ignored).

8.5.3 Package Renaming Declarations

Insert after paragraph 3:

The renamed entity shall be a package.

the new paragraph:

If the *package_*name of a package_renaming_declaration denotes a limited view of a package *P*, then a name that denotes the package_renaming_declaration shall occur only within the immediate scope of the renaming or the scope of a with_clause that mentions the package *P* or, if *P* is a nested package, the innermost library package enclosing *P*.

Insert after paragraph 4:

A package_renaming_declaration declares a new view of the renamed package.

the new paragraph:

At places where the declaration of the limited view of the renamed package is visible, a name that denotes the package_renaming_declaration denotes a limited view of the package (see 10.1.1).

8.5.4 Subprogram Renaming Declarations

Replace paragraph 2:

subprogram_renaming_declaration ::= subprogram_specification **renames** *callable_entity_*name;

by:

subprogram_renaming_declaration ::=
  [overriding_indicator]
  subprogram_specification **renames** *callable_entity_*name;

Insert after paragraph 4:

The profile of a renaming-as-declaration shall be mode-conformant with that of the renamed callable entity.

the new paragraphs:

For a parameter or result subtype of the subprogram_specification that has an explicit null_exclusion:

- if the *callable_entity*_name denotes a generic formal subprogram of a generic unit *G*, and the subprogram_renaming_declaration occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result subtype of the formal subprogram of *G* shall have a null_exclusion;
- otherwise, the subtype of the corresponding parameter or result type of the renamed callable entity shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Insert after paragraph 5:

The profile of a renaming-as-body shall be subtype-conformant with that of the renamed callable entity, and shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the profile shall be mode-conformant with that of the renamed callable entity and the subprogram it declares takes its convention from the renamed subprogram; otherwise, the profile shall be subtype-conformant with that of the renamed callable entity and the convention of the renamed subprogram shall not be Intrinsic. A renaming-as-body is illegal if the declaration occurs before the subprogram whose declaration it completes is frozen, and the renaming renames the subprogram itself, through one or more subprogram renaming declarations, none of whose subprograms has been frozen.

the new paragraph:

The *callable_entity*_name of a renaming shall not denote a subprogram that requires overriding (see 3.9.3).

The *callable_entity*_name of a renaming-as-body shall not denote an abstract subprogram.

8.6 The Context of Overload Resolution

Replace paragraph 17:

If a usage name appears within the declarative region of a type_declaration and denotes that same type_declaration, then it denotes the *current instance* of the type (rather than the type itself). The current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name.

by:

If a usage name appears within the declarative region of a type_declaration and denotes that same type_declaration, then it denotes the *current instance* of the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. This rule does not apply if the usage name appears within the subtype_mark of an access_definition for an access-to-object type, or within the subtype of a parameter or result of an access-to-subprogram type.

Replace paragraph 20:

The *expected type* for a given expression, name, or other construct determines, according to the *type resolution rules* given below, the types considered for the construct during overload resolution. The type resolution rules provide support for class-wide programming, universal numeric literals, dispatching operations, and anonymous access types:

by:

The *expected type* for a given expression, name, or other construct determines, according to the *type resolution rules* given below, the types considered for the construct during overload resolution. The type resolution rules provide support for class-wide programming, universal literals, dispatching operations, and anonymous access types:

Replace paragraph 25:

- when *T* is an anonymous access type (see 3.10) with designated type *D*, to an access-to-variable type whose designated type is *D*'Class or is covered by *D*.

by:

- when *T* is a specific anonymous access-to-object type (see 3.10) with designated type *D*, to an access-to-object type whose designated type is *D*'Class or is covered by *D*; or
- when *T* is an anonymous access-to-subprogram type (see 3.10), to an access-to-subprogram type whose designated profile is type-conformant with that of *T*.

Replace paragraph 27:

When the expected type for a construct is required to be a *single* type in a given class, the type expected for the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class; the type of the construct is then this single expected type. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a type_conversion.

by:

When a construct is one that requires that its expected type be a *single* type in a given class, the type of the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a type_conversion.

Section 9: Tasks and Synchronization

9.1 Task Units and Task Objects

Replace paragraph 2:

task_type_declaration ::=
  task type defining_identifier [known_discriminant_part] [**is** task_definition];

by:

task_type_declaration ::=
  task type defining_identifier [known_discriminant_part] [**is**
    [**new** interface_list **with**]
    task_definition];

Replace paragraph 3:

single_task_declaration ::=
  task defining_identifier [**is** task_definition];

by:

single_task_declaration ::=
  task defining_identifier [**is**
    [**new** interface_list **with**]
    task_definition];

Delete paragraph 8:

Legality Rules

A task declaration requires a completion, which shall be a task_body, and every task_body shall be the completion of some task declaration.

Insert after paragraph 9.1:

For a task declaration without a task_definition, a task_definition without task_items is assumed.

the new paragraphs:

For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming task entry.

Legality Rules

A task declaration requires a completion, which shall be a task_body, and every task_body shall be the completion of some task declaration.

Each interface_subtype_mark of an interface_list appearing within a task declaration shall denote a limited interface type that is not a protected interface.

The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the first parameter of the subprogram is of the task type or is an access parameter designating the task type.

For each primitive subprogram inherited by the type declared by a task declaration, at most one of the following shall apply:

- the inherited subprogram is overridden with a primitive subprogram of the task type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or

- the inherited subprogram is implemented by a single entry of the task type; in which case its prefixed view profile shall be subtype conformant with that of the task entry.

If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Replace paragraph 19:

2 Within the declaration or body of a task unit, the name of the task unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a subtype_mark).

by:

2 Other than in an access_definition, the name of a task unit within the declaration or body of the task unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a subtype_mark).

Replace paragraph 21:

4 A task type is a limited type (see 7.5), and hence has neither an assignment operation nor predefined equality operators. 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. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7).

by:

4 A task type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators. 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. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7.1).

Replace paragraph 24:

```
task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;
```

by:

```
task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
      new Serial_Device with  -- see 3.9.4
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;
```

9.2 Task Execution - Task Activation

Replace paragraph 2:

A task object (which represents one task) can be created either as part of the elaboration of an object_declaration occurring immediately within some declarative region, or as part of the evaluation of an allocator. All tasks created by the elaboration of object_declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together. The activation of a task is associated with the innermost allocator or object_declaration that is responsible for its creation.

by:

A task object (which represents one task) can be a part of a stand-alone object, of an object created by an allocator, or of an anonymous object of a limited type, or a coextension of one of these. All tasks that are part or coextensions of any of the stand-alone objects created by the elaboration of object_declarations (or generic_associations of formal objects of mode **in**) of a single declarative region are activated together. All tasks that are part or coextensions of a single object that is not a stand-alone object are activated together.

Replace paragraph 3:

For tasks created by the elaboration of object_declarations of a given declarative region, the activations are initiated within the context of the handled_sequence_of_statements (and its associated exception_handlers if any — see 11.2), just prior to executing the statements of the _sequence. For a package without an explicit body or an explicit handled_sequence_of_statements, an implicit body or an implicit null_statement is assumed, as defined in 7.2.

by:

For the tasks of a given declarative region, the activations are initiated within the context of the handled_sequence_of_statements (and its associated exception_handlers if any — see 11.2), just prior to executing the statements of the handled_sequence_of_statements. For a package without an explicit body or an explicit handled_sequence_of_statements, an implicit body or an implicit null_statement is assumed, as defined in 7.2.

Replace paragraph 4:

For tasks created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, after completing any initialization for the object created by the allocator, and prior to returning the new access value.

by:

For tasks that are part or coextensions of a single object that is not a stand-alone object, activations are initiated after completing any initialization of the outermost object enclosing these tasks, prior to performing any other operation on the outermost object. In particular, for tasks that are part or coextensions of the object created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, prior to returning the new access value. For tasks that are part or coextensions of an object that is the result of a function call, the activations are not initiated until after the function returns.

9.3 Task Dependence - Termination of Tasks

Insert after paragraph 3:

- If the task is created by the elaboration of an object_declaration, it depends on each master that includes this elaboration.

the new paragraph:

- Otherwise, the task depends on the master of the outermost object of which it is a part (as determined by the accessibility level of that object — see 3.10.2 and 7.6.1), as well as on any master whose execution includes that of the master of the outermost object.

Replace paragraph 7:

- The task depends on some completed master;

by:

- The task depends on some completed master; and

9.4 Protected Units and Protected Objects

Replace paragraph 2:

```
protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part] [is protected_definition];
```

by:

```
protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part] is
    [new interface_list with]
    protected_definition;
```

Replace paragraph 3:

```
single_protected_declaration ::=
```

protected defining_identifier **is** protected_definition;

by:

```
single_protected_declaration ::=
  protected defining_identifier is
    [new interface_list with]
    protected_definition;
```

Delete paragraph 10:

Legality Rules

A protected declaration requires a completion, which shall be a protected_body, and every protected_body shall be the completion of some protected declaration.

Insert after paragraph 11:

A protected_definition defines a protected type and its first subtype. The list of protected_operation_declarations of a protected_definition, together with the known_discriminant_part, if any, is called the visible part of the protected unit. The optional list of protected_element_declarations after the reserved word **private** is called the private part of the protected unit.

the new paragraphs:

For a protected declaration with an interface_list, the protected type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry.

Legality Rules

A protected declaration requires a completion, which shall be a protected_body, and every protected_body shall be the completion of some protected declaration.

Each interface_subtype_mark of an interface_list appearing within a protected declaration shall denote a limited interface type that is not a task interface.

The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type.

For each primitive subprogram inherited by the type declared by a protected declaration, at most one of the following shall apply:

- the inherited subprogram is overridden with a primitive subprogram of the protected type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or
- the inherited subprogram is implemented by a protected subprogram or single entry of the protected type, in which case its prefixed view profile shall be subtype conformant with that of the protected subprogram or entry.

If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

If an inherited subprogram is implemented by a protected procedure or an entry, then the first parameter of the inherited subprogram shall be of mode **out** or **in out**, or an access-to-variable parameter.

If a protected subprogram declaration has an overriding_indicator, then at the point of the declaration:

- if the overriding_indicator is **overriding**, then the subprogram shall implement an inherited subprogram;
- if the overriding_indicator is **not overriding**, then the subprogram shall not implement any inherited subprogram.

Insert after paragraph 20:

As the first step of the *finalization* of a protected object, each call remaining on any entry queue of the object is removed from its queue and Program_Error is raised at the place of the corresponding entry_call_statement.

the new paragraph:

Bounded (Run-Time) Errors

It is a bounded error to call an entry or subprogram of a protected object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task queued forever.

Replace paragraph 21:

13 Within the declaration or body of a protected unit, the name of the protected unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a subtype_mark).

by:

13 Within the declaration or body of a protected unit other than in an access_definition, the name of the protected unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a subtype_mark).

Replace paragraph 23:

15 A protected type is a limited type (see 7.5), and hence has neither an assignment operation nor predefined equality operators.

by:

15 A protected type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators.

9.5 Intertask Communication

Insert after paragraph 7:

A corresponding definition of target object applies to a requeue_statement (see 9.5.4), with a corresponding distinction between an *internal requeue* and an *external requeue*.

the new paragraph:

The view of the target protected object associated with a call of a protected procedure or entry shall be a variable.

9.5.1 Protected Subprograms and Protected Actions

Insert after paragraph 22:

21 From within a protected action, an internal call on a protected subprogram, or an external call on a protected subprogram with a different target object is not considered a potentially blocking operation.

the new paragraph:

22 The pragma Detect_Blocking may be used to ensure that all executions of potentially blocking operations during a protected action raise Program_Error. See H.5.

9.5.2 Entries and Accept Statements

Replace paragraph 2:

```
entry_declaration ::=
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile;
```

by:

```
entry_declaration ::=
  [overriding_indicator]
```

entry defining_identifier [(discrete_subtype_definition)] parameter_profile;

Insert after paragraph 10:

An entry_declaration is allowed only in a protected or task declaration.

the new paragraph:

An overriding_indicator is not allowed in an entry_declaration that includes a discrete_subtype_definition.

Insert after paragraph 13:

An entry_declaration in a task declaration shall not contain a specification for an access parameter (see 3.10).

the new paragraphs:

If an entry_declaration has an overriding_indicator, then at the point of the declaration:

- if the overriding_indicator is **overriding**, then the entry shall implement an inherited subprogram;
- if the overriding_indicator is **not overriding**, then the entry shall not implement any inherited subprogram.

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Replace paragraph 29:

24 A return_statement (see 6.5) or a requeue_statement (see 9.5.4) may be used to complete the execution of an accept_statement or an entry_body.

by:

24 A return statement (see 6.5) or a requeue_statement (see 9.5.4) may be used to complete the execution of an accept_statement or an entry_body.

9.6 Delay Statements, Duration, and Time

Replace paragraph 11:

```
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;
```

by:

```
subtype Year_Number  is Integer range 1901 .. 2399;
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;
```

Replace paragraph 24:

The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined timezone; 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.

by:

The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined time zone; 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.

9.6.1 Formatting, Time Zones, and other operations for Time

Insert new clause:

Static Semantics

The following language-defined library packages exist:

```
package Ada.Calendar.Time_Zones is

    -- Time zone manipulation:

    type Time_Offset is range -28*60 .. 28*60;

    Unknown_Zone_Error : exception;

    function UTC_Time_Offset (Date : Time := Clock) return Time_Offset;

end Ada.Calendar.Time_Zones;


package Ada.Calendar.Arithmetic is

    -- Arithmetic on days:

    type Day_Count is range
      -366*(1+Year_Number'Last - Year_Number'First)
      ..
      366*(1+Year_Number'Last - Year_Number'First);

    subtype Leap_Seconds_Count is Integer range -2047 .. 2047;

    procedure Difference (Left, Right : in Time;
                          Days : out Day_Count;
                          Seconds : out Duration;
                          Leap_Seconds : out Leap_Seconds_Count);

    function "+" (Left : Time; Right : Day_Count) return Time;
    function "+" (Left : Day_Count; Right : Time) return Time;
    function "-" (Left : Time; Right : Day_Count) return Time;
    function "-" (Left, Right : Time) return Day_Count;

end Ada.Calendar.Arithmetic;

with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is

    -- Day of the week:

    type Day_Name is (Monday, Tuesday, Wednesday, Thursday,
        Friday, Saturday, Sunday);

    function Day_of_Week (Date : Time) return Day_Name;

    -- Hours:Minutes:Seconds access:

    subtype Hour_Number         is Natural range 0 .. 23;
    subtype Minute_Number       is Natural range 0 .. 59;
    subtype Second_Number       is Natural range 0 .. 59;
    subtype Second_Duration     is Day_Duration range 0.0 .. 1.0;

    function Year       (Date : Time;
                         Time_Zone  : Time_Zones.Time_Offset := 0)
                            return Year_Number;

    function Month      (Date : Time;
                         Time_Zone  : Time_Zones.Time_Offset := 0)
                            return Month_Number;
```

```
function Day        (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Day_Number;

function Hour       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Hour_Number;

function Minute     (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Minute_Number;

function Second     (Date : Time)
                        return Second_Number;

function Sub_Second (Date : Time)
                        return Second_Duration;

function Seconds_Of (Hour   :  Hour_Number;
                     Minute : Minute_Number;
                     Second : Second_Number := 0;
                     Sub_Second : Second_Duration := 0.0)
    return Day_Duration;

procedure Split (Seconds    : in Day_Duration;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration);

function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Hour       : Hour_Number;
                  Minute     : Minute_Number;
                  Second     : Second_Number;
                  Sub_Second : Second_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                         return Time;

function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                         return Time;

procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);

procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
```

```
                    Sub_Second : out Second_Duration;
                    Leap_Second: out Boolean;
                    Time_Zone  : in Time_Zones.Time_Offset := 0);

    procedure Split (Date       : in Time;
                    Year       : out Year_Number;
                    Month      : out Month_Number;
                    Day        : out Day_Number;
                    Seconds    : out Day_Duration;
                    Leap_Second: out Boolean;
                    Time_Zone  : in Time_Zones.Time_Offset := 0);

    -- Simple image and value:
    function Image (Date : Time;
                    Include_Time_Fraction : Boolean := False;
                    Time_Zone  : Time_Zones.Time_Offset := 0) return String;

    function Value (Date : String;
                    Time_Zone  : Time_Zones.Time_Offset := 0) return Time;

    function Image (Elapsed_Time : Duration;
                    Include_Time_Fraction : Boolean := False) return String;

    function Value (Elapsed_Time : String) return Duration;

end Ada.Calendar.Formatting;
```

Type Time_Offset represents the number of minutes difference between the implementation-defined time zone used by Calendar and another time zone.

```
function UTC_Time_Offset (Date : Time := Clock) return Time_Offset;
```

Returns, as a number of minutes, the difference between the implementation-defined time zone of Calendar, and UTC time, at the time Date. If the time zone of the Calendar implementation is unknown, then Unknown_Zone_Error is raised.

```
procedure Difference (Left, Right : in Time;
                      Days : out Day_Count;
                      Seconds : out Duration;
                      Leap_Seconds : out Leap_Seconds_Count);
```

Returns the difference between Left and Right. Days is the number of days of difference, Seconds is the remainder seconds of difference excluding leap seconds, and Leap_Seconds is the number of leap seconds. If Left < Right, then Seconds <= 0.0, Days <= 0, and Leap_Seconds <= 0. Otherwise, all values are nonnegative. The absolute value of Seconds is always less than 86_400.0. For the returned values, if Days = 0, then Seconds + Duration(Leap_Seconds) = Calendar."–" (Left, Right).

```
function "+" (Left : Time; Right : Day_Count) return Time;
function "+" (Left : Day_Count; Right : Time) return Time;
```

Adds a number of days to a time value. Time_Error is raised if the result is not representable as a value of type Time.

```
function "-" (Left : Time; Right : Day_Count) return Time;
```

Subtracts a number of days from a time value. Time_Error is raised if the result is not representable as a value of type Time.

```
function "-" (Left, Right : Time) return Day_Count;
```

Subtracts two time values, and returns the number of days between them. This is the same value that Difference would return in Days.

```
function Day_of_Week (Date : Time) return Day_Name;
```

Returns the day of the week for Time. This is based on the Year, Month, and Day values of Time.

```
function Year       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
```

```
                        return Year_Number;
```

Returns the year for Date, as appropriate for the specified time zone offset.

```
function Month      (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Month_Number;
```

Returns the month for Date, as appropriate for the specified time zone offset.

```
function Day        (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Day_Number;
```

Returns the day number for Date, as appropriate for the specified time zone offset.

```
function Hour       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Hour_Number;
```

Returns the hour for Date, as appropriate for the specified time zone offset.

```
function Minute     (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Minute_Number;
```

Returns the minute within the hour for Date, as appropriate for the specified time zone offset.

```
function Second     (Date : Time)
                        return Second_Number;
```

Returns the second within the hour and minute for Date.

```
function Sub_Second (Date : Time)
                        return Second_Duration;
```

Returns the fraction of second for Date (this has the same accuracy as Day_Duration). The value returned is always less than 1.0.

```
function Seconds_Of (Hour   : Hour_Number;
                     Minute : Minute_Number;
                     Second : Second_Number := 0;
                     Sub_Second : Second_Duration := 0.0)
    return Day_Duration;
```

Returns a Day_Duration value for the combination of the given Hour, Minute, Second, and Sub_Second. This value can be used in Calendar.Time_Of as well as the argument to Calendar."+" and Calendar."–". If Seconds_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Seconds_Of for the next second with a Sub_Second value of 0.0.

```
procedure Split (Seconds    : in Day_Duration;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration);
```

Splits Seconds into Hour, Minute, Second and Sub_Second in such a way that the resulting values all belong to their respective subtypes. The value returned in the Sub_Second parameter is always less than 1.0.

```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Hour       : Hour_Number;
                  Minute     : Minute_Number;
                  Second     : Second_Number;
                  Sub_Second : Second_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                         return Time;
```

If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second

later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Time_Of for the next second with a Sub_Second value of 0.0.

```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                         return Time;
```

If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a Seconds value of 0.0.

```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

Splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset. The value returned in the Sub_Second parameter is always less than 1.0.

```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Sub_Second parameter is always less than 1.0.

```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Seconds    : out Day_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Seconds), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Seconds parameter is always less than 86_400.0.

```
function Image (Date : Time;
                Include_Time_Fraction : Boolean := False;
                Time_Zone  : Time_Zones.Time_Offset := 0) return String;
```

Returns a string form of the Date relative to the given Time_Zone. The format is "Year-Month-Day Hour:Minute:Second", where the Year is a 4-digit value, and all others are 2-digit values, of the functions defined in Calendar and Calendar.Formatting, including a leading zero, if needed. The separators between the values are a minus, another minus, a colon, and a single space between the Day and Hour. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value.

```
function Value (Date : String;
                Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
```

Returns a Time value for the image given as Date, relative to the given time zone. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Time value.

```
function Image (Elapsed_Time : Duration;
                Include_Time_Fraction : Boolean := False) return String;
```

Returns a string form of the Elapsed_Time. The format is "Hour:Minute:Second", where all values are 2-digit values, including a leading zero, if needed. The separators between the values are colons. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value. If Elapsed_Time < 0.0, the result is Image (**abs** Elapsed_Time, Include_Time_Fraction) prefixed with a minus sign. If **abs** Elapsed_Time represents 100 hours or more, the result is implementation-defined.

```
function Value (Elapsed_Time : String) return Duration;
```

Returns a Duration value for the image given as Elapsed_Time. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Duration value.

Implementation Advice

An implementation should support leap seconds if the target system supports them. If leap seconds are not supported, Difference should return zero for Leap_Seconds, Split should return False for Leap_Second, and Time_Of should raise Time_Error if Leap_Second is True.

NOTES

36 The implementation-defined time zone of package Calendar may, but need not, be the local time zone. UTC_Time_Offset always returns the difference relative to the implementation-defined time zone of package Calendar. If UTC_Time_Offset does not raise Unknown_Zone_Error, UTC time can be safely calculated (within the accuracy of the underlying time-base).

37 Calling Split on the results of subtracting Duration(UTC_Time_Offset*60) from Clock provides the components (hours, minutes, and so on) of the UTC time. In the United States, for example, UTC_Time_Offset will generally be negative.

9.7.2 Timed Entry Calls

Replace paragraph 1:

A timed_entry_call issues an entry call that is cancelled if the call (or a requeue-with-abort of the call) is not selected before the expiration time is reached.

by:

A timed_entry_call issues an entry call that is cancelled if the call (or a requeue-with-abort of the call) is not selected before the expiration time is reached. A procedure call may appear rather than an entry call for cases where the procedure might be implemented by an entry.

Replace paragraph 3:

```
entry_call_alternative ::=
  entry_call_statement [sequence_of_statements]
```

by:

```
entry_call_alternative ::=
  procedure_or_entry_call [sequence_of_statements]

procedure_or_entry_call ::=
```

procedure_call_statement | entry_call_statement

Legality Rules

If a procedure_call_statement is used for a procedure_or_entry_call, the *procedure_*name or *procedure_*prefix of the procedure_call_statement shall statically denote an entry renamed as a procedure or (a view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter (see 3.9.2).

Static Semantics

If a procedure_call_statement is used for a procedure_or_entry_call, and the procedure is implemented by an entry, then the *procedure_*name, or *procedure_*prefix and possibly the first parameter of the procedure_call_statement, determine the target object of the call and the entry to be called.

Replace paragraph 4:

For the execution of a timed_entry_call, the *entry_*name and the actual parameters are evaluated, as for a simple entry call (see 9.5.3). The expiration time (see 9.6) for the call is determined by evaluating the *delay_*expression of the delay_alternative; the entry call is then issued.

by:

For the execution of a timed_entry_call, the *entry_*name, *procedure_*name, or *procedure_*prefix, and any actual parameters are evaluated, as for a simple entry call (see 9.5.3) or procedure call (see 6.4). The expiration time (see 9.6) for the call is determined by evaluating the *delay_*expression of the delay_alternative. If the call is an entry call or a call on a procedure implemented by an entry, the entry call is then issued. Otherwise, the call proceeds as described in 6.4 for a procedure call, followed by the sequence_of_statements of the entry_call_alternative; the sequence_of_statements of the delay_alternative is ignored.

9.7.3 Conditional Entry Calls

Replace paragraph 1:

A conditional_entry_call issues an entry call that is then cancelled if it is not selected immediately (or if a requeue-with-abort of the call is not selected immediately).

by:

A conditional_entry_call issues an entry call that is then cancelled if it is not selected immediately (or if a requeue-with-abort of the call is not selected immediately). A procedure call may appear rather than an entry call for cases where the procedure might be implemented by an entry.

9.7.4 Asynchronous Transfer of Control

Replace paragraph 4:

triggering_statement ::= entry_call_statement | delay_statement

by:

triggering_statement ::= procedure_or_entry_call | delay_statement

Replace paragraph 6:

For the execution of an asynchronous_select whose triggering_statement is an entry_call_statement, the *entry_*name and actual parameters are evaluated as for a simple entry call (see 9.5.3), and the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started.

by:

For the execution of an asynchronous_select whose triggering_statement is a procedure_or_entry_call, the *entry_*name, *procedure_*name, or *procedure_*prefix, and actual parameters are evaluated as for a simple entry call (see 9.5.3) or procedure call (see 6.4). If the call is an entry call or a call on a procedure implemented by an entry, the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started. If the call is on a

procedure that is not implemented by an entry, the call proceeds as described in 6.4, followed by the sequence_of_statements of the triggering_alternative; the abortable_part is never started.

9.11 Example of Tasking and Synchronization

Replace paragraph 3:

```
task body Producer is
   Char : Character;
begin
   loop
      ... -- produce the next character Char
      Buffer.Write(Char);
      exit when Char = ASCII.EOT;
   end loop;
end Producer;
```

by:

```
task body Producer is
   Person : Person_Name; -- see 3.10.1
begin
   loop
      ... -- simulate arrival of the next customer
      Buffer.Append_Wait(Person);
      exit when Person = null;
   end loop;
end Producer;
```

Replace paragraph 6:

```
task body Consumer is
   Char : Character;
begin
   loop
      Buffer.Read(Char);
      exit when Char = ASCII.EOT;
      ... -- consume the character Char
   end loop;
end Consumer;
```

by:

```
task body Consumer is
   Person : Person_Name;
begin
   loop
      Buffer.Remove_First_Wait(Person);
      exit when Person = null;
      ... -- simulate serving a customer
   end loop;
end Consumer;
```

Replace paragraph 7:

The buffer object contains an internal pool of characters managed 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.

by:

The buffer object contains an internal array of person names managed in a round-robin fashion. The array has two indices, an In_Index denoting the index for the next input person name and an Out_Index denoting the index for the next output person name.

The Buffer is defined as an extension of the Synchronized_Queue interface (see 3.9.4), and as such promises to implement the abstraction defined by that interface. By doing so, the Buffer can be passed to the Transfer class-wide operation defined for objects of a type covered by Queue'Class.

Replace paragraph 8:

```
protected Buffer is
   entry Read (C : out Character);
   entry Write(C : in  Character);
private
   Pool      : String(1 .. 100);
   Count     : Natural := 0;
   In_Index, Out_Index : Positive := 1;
end Buffer;
```

by:

```
protected Buffer is new Synchronized_Queue with  -- see 3.9.4
   entry Append_Wait(Person : in Person_Name);
   entry Remove_First_Wait(Person : out Person_Name);
   function Cur_Count return Natural;
   function Max_Count return Natural;
   procedure Append(Person : in Person_Name);
   procedure Remove_First(Person : out Person_Name);
private
   Pool      : Person_Name_Array(1 .. 100);
   Count     : Natural := 0;
   In_Index, Out_Index : Positive := 1;
end Buffer;
```

Replace paragraph 9:

```
protected body Buffer is
   entry Write(C : in Character)
      when Count < Pool'Length is
   begin
      Pool(In_Index) := C;
      In_Index := (In_Index mod Pool'Length) + 1;
      Count    := Count + 1;
   end Write;
```

by:

```
protected body Buffer is
   entry Append_Wait(Person : in Person_Name)
      when Count < Pool'Length is
   begin
      Append(Person);
   end Append_Wait;

   procedure Append(Person : in Person_Name) is
   begin
      if Count = Pool'Length then
         raise Queue_Error with "Buffer Full";  -- see 11.3
      end if;
      Pool(In_Index) := Person;
      In_Index       := (In_Index mod Pool'Length) + 1;
      Count          := Count + 1;
   end Append;
```

Replace paragraph 10:

```
   entry Read(C : out Character)
      when Count > 0 is
   begin
      C := Pool(Out_Index);
      Out_Index := (Out_Index mod Pool'Length) + 1;
      Count     := Count - 1;
   end Read;
```

```
      end Buffer;
```

by:

```
         entry Remove_First_Wait(Person : out Person_Name)
            when Count > 0 is
         begin
            Remove_First(Person);
         end Remove_First_Wait;

         procedure Remove_First(Person : out Person_Name) is
         begin
            if Count = 0 then
               raise Queue_Error with "Buffer Empty"; -- see 11.3
            end if;
            Person    := Pool(Out_Index);
            Out_Index := (Out_Index mod Pool'Length) + 1;
            Count     := Count - 1;
         end Remove_First;

         function Cur_Count return Natural is
         begin
             return Buffer.Count;
         end Cur_Count;

         function Max_Count return Natural is
         begin
             return Pool'Length;
         end Max_Count;
      end Buffer;
```

Section 10: Program Structure and Compilation Issues

10.1.1 Compilation Units - Library Units

Insert after paragraph 8:

parent_unit_name ::= name

the new paragraph:

An overriding_indicator is not allowed in a subprogram_declaration, generic_instantiation, or subprogram_renaming_declaration that declares a library unit.

Insert after paragraph 12:

A library_unit_declaration or a library_unit_renaming_declaration is *private* if the declaration is immediately preceded by the reserved word **private**; it is otherwise *public*. A library unit is private or public according to its declaration. The *public descendants* of a library unit are the library unit itself, and the public descendants of its public children. Its other descendants are *private descendants*.

the new paragraphs:

For each library package_declaration in the environment, there is an implicit declaration of a *limited view* of that library package. The limited view of a package contains:

- For each nested package_declaration, a declaration of the limited view of that package, with the same defining_program_unit_name.
- For each type_declaration in the visible part, an incomplete view of the type; if the type_declaration is tagged, then the view is a tagged incomplete view.

The limited view of a library package_declaration is private if that library package_declaration is immediately preceded by the reserved word **private**.

There is no syntax for declaring limited views of packages, because they are always implicit. The implicit declaration of a limited view of a library package is not the declaration of a library unit (the library package_declaration is); nonetheless, it is a library_item. The implicit declaration of the limited view of a library package forms an (implicit) compilation unit whose context_clause is empty.

A library package_declaration is the completion of the declaration of its limited view.

Replace paragraph 15:

A parent_unit_name (which can be used within a defining_program_unit_name of a library_item and in the **separate** clause of a subunit), and each of its prefixes, shall not denote a renaming_declaration. On the other hand, a name that denotes a library_unit_renaming_declaration is allowed in a with_clause and other places where the name of a library unit is allowed.

by:

A parent_unit_name (which can be used within a defining_program_unit_name of a library_item and in the **separate** clause of a subunit), and each of its prefixes, shall not denote a renaming_declaration. On the other hand, a name that denotes a library_unit_renaming_declaration is allowed in a nonlimited_with_clause and other places where the name of a library unit is allowed.

Replace paragraph 19:

For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent. This declaration is visible only within the scope of a with_clause that mentions the child generic unit.

by:

For each child *C* of some parent generic package *P*, there is a corresponding declaration *C* nested immediately within each instance of *P*. For the purposes of this rule, if a child *C* itself has a child *D*, each corresponding declaration for *C* has a

corresponding child *D*. The corresponding declaration for a child within an instance is visible only within the scope of a with_clause that mentions the (original) child generic unit.

Replace paragraph 26:

A library_item depends semantically upon its parent declaration. A subunit depends semantically upon its parent body. A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. In addition, if a given compilation unit contains an attribute_reference of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit. The semantic dependence relationship is transitive.

by:

A library_item depends semantically upon its parent declaration. A subunit depends semantically upon its parent body. A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. The declaration of the limited view of a library package depends semantically upon the declaration of the limited view of its parent. The declaration of a library package depends semantically upon the declaration of its limited view. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. In addition, if a given compilation unit contains an attribute_reference of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit. The semantic dependence relationship is transitive.

Dynamic Semantics

The elaboration of the declaration of the limited view of a package has no effect.

10.1.2 Context Clauses - With Clauses

Replace paragraph 4:

with_clause ::= **with** *library_unit_*name {, *library_unit_*name};

by:

with_clause ::= limited_with_clause | nonlimited_with_clause
limited_with_clause ::= **limited** [**private**] **with** *library_unit_*name {, *library_unit_*name};
nonlimited_with_clause ::= [**private**] **with** *library_unit_*name {, *library_unit_*name};

Replace paragraph 6:

A library_item is *mentioned* in a with_clause if it is denoted by a *library_unit_*name or a prefix in the with_clause.

by:

A library_item (and the corresponding library unit) is *named* in a with_clause if it is denoted by a *library_unit_*name in the with_clause. A library_item (and the corresponding library unit) is *mentioned* in a with_clause if it is named in the with_clause or if it is denoted by a prefix in the with_clause.

Replace paragraph 8:

If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be either the declaration of a private descendant of that library unit or the body or a subunit of a (public or private) descendant of that library unit.

by:

If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be one of:

- the declaration, body, or subunit of a private descendant of that library unit;
- the body or subunit of a public descendant of that library unit, but not a subprogram body acting as a subprogram declaration (see 10.1.4); or

- the declaration of a public descendant of that library unit, in which case the with_clause shall include the reserved word **private**.

A name denoting a library item that is visible only due to being mentioned in one or more with_clauses that include the reserved word **private** shall appear only within

- a private part;
- a body, but not within the subprogram_specification of a library subprogram body;
- a private descendant of the unit on which one of these with_clauses appear; or
- a pragma within a context clause.

A library_item mentioned in a limited_with_clause shall be the implicit declaration of the limited view of a library package, not the declaration of a subprogram, generic unit, generic instance, or a renaming.

A limited_with_clause shall not appear on a library_unit_body, subunit, or library_unit_renaming_declaration.

A limited_with_clause that names a library package shall not appear:

- in the context_clause for the explicit declaration of the named library package;
- in the same context_clause as, or within the scope of, a nonlimited_with_clause that mentions the same library package; or
- in the same context_clause as, or within the scope of, a use_clause that names an entity declared within the declarative region of the library package.

Examples

```
package Office is
end Office;

with Ada.Strings.Unbounded;
package Office.Locations is
   type Location is new Ada.Strings.Unbounded.Unbounded_String;
end Office.Locations;

limited with Office.Departments;  -- types are incomplete
private with Office.Locations;    -- only visible in private part
package Office.Employees is
   type Employee is private;

   function Dept_Of(Emp : Employee) return access Departments.Department;
   procedure Assign_Dept(Emp  : in out Employee;
                         Dept : access Departments.Department);

   ...
private
   type Employee is
      record
         Dept : access Departments.Department;
         Loc : Locations.Location;
         ...
      end record;
end Office.Employees;

limited with Office.Employees;
package Office.Departments is
   type Department is private;

   function Manager_Of(Dept : Department) return access Employees.Employee;
   procedure Assign_Manager(Dept : in out Department;
                            Mgr  : access Employees.Employee);
   ...
end Office.Departments;
```

The limited_with_clause may be used to support mutually dependent abstractions that are split across multiple packages. In this case, an employee is assigned to a department, and a department has a manager who is an employee. If a with_clause with the reserved word **private** appears on one library unit and mentions a second library unit, it provides visibility to the second library unit, but restricts that visibility to the private part and body of the first unit. The compiler checks that no use is made of the second unit in the visible part of the first unit.

Replace paragraph 9:

A library_item mentioned in a with_clause of a compilation unit is visible within the compilation unit and hence acts just like an ordinary declaration. Thus, within a compilation unit that mentions its declaration, 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 generic library unit can be declared. If a child of a parent generic package is mentioned in a with_clause, then the corresponding declaration nested within each visible instance is visible within the compilation unit.

by:

A library_item mentioned in a nonlimited_with_clause of a compilation unit is visible within the compilation unit and hence acts just like an ordinary declaration. Thus, within a compilation unit that mentions its declaration, 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 generic library unit can be declared. If a child of a parent generic package is mentioned in a nonlimited_with_clause, then the corresponding declaration nested within each visible instance is visible within the compilation unit. Similarly, a library_item mentioned in a limited_with_clause of a compilation unit is visible within the compilation unit and thus can be used to form expanded names.

10.1.3 Subunits of Compilation Units

Replace paragraph 3:

subprogram_body_stub ::= subprogram_specification **is separate**;

by:

```
subprogram_body_stub ::=
   [overriding_indicator]
   subprogram_specification is separate;
```

Replace paragraph 8:

The *parent body* of a subunit is the body of the program unit denoted by its parent_unit_name. The term *subunit* is used to refer to a subunit and also to the proper_body of a subunit.

by:

The *parent body* of a subunit is the body of the program unit denoted by its parent_unit_name. The term *subunit* is used to refer to a subunit and also to the proper_body of a subunit. The *subunits of a program unit* include any subunit that names that program unit as its parent, as well as any subunit that names such a subunit as its parent (recursively).

10.1.4 The Compilation Process

Replace paragraph 3:

The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined.

by:

The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined. The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment are implementation defined.

Replace paragraph 6:

The implementation may require that a compilation unit be legal before inserting it into the environment.

by:

The implementation may require that a compilation unit be legal before it can be mentioned in a limited_with_clause or it can be inserted into the environment.

Replace paragraph 7:

When a compilation unit that declares or renames a library unit is added to the environment, the implementation may remove from the environment any preexisting library_item with the same defining_program_unit_name. When a compilation unit that is a subunit or the body of a library unit is added to the environment, the implementation may remove from the environment any preexisting version of the same compilation unit. When a given compilation unit is removed from the environment, the implementation may also remove any compilation unit that depends semantically upon the given one. If the given compilation unit contains the body of a subprogram to which a pragma Inline applies, the implementation may also remove any compilation unit containing a call to that subprogram.

by:

When a compilation unit that declares or renames a library unit is added to the environment, the implementation may remove from the environment any preexisting library_item or subunit with the same full expanded name. When a compilation unit that is a subunit or the body of a library unit is added to the environment, the implementation may remove from the environment any preexisting version of the same compilation unit. When a compilation unit that contains a body_stub is added to the environment, the implementation may remove any preexisting library_item or subunit with the same full expanded name as the body_stub. When a given compilation unit is removed from the environment, the implementation may also remove any compilation unit that depends semantically upon the given one. If the given compilation unit contains the body of a subprogram to which a pragma Inline applies, the implementation may also remove any compilation unit containing a call to that subprogram.

10.1.5 Pragmas and Program Units

Replace paragraph 9:

An implementation may place restrictions on configuration pragmas, so long as it allows them when the environment contains no library_items other than those of the predefined environment.

by:

An implementation may require that configuration pragmas that select partition-wide or system-wide options be compiled when the environment contains no library_items other than those of the predefined environment. In this case, the implementation shall still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options.

10.1.6 Environment-Level Visibility Rules

Replace paragraph 2:

Within the parent_unit_name at the beginning of a library_item, and within a with_clause, the only declarations that are visible are those that are library_items of the environment, and the only declarations that are directly visible are those that are root library_items of the environment. Notwithstanding the rules of 4.1.3, an expanded name in a with_clause may consist of a prefix that denotes a generic package and a selector_name that denotes a child of that generic package. (The child is necessarily a generic unit; see 10.1.1.)

by:

Within the parent_unit_name at the beginning of an explicit library_item, and within a nonlimited_with_clause, the only declarations that are visible are those that are explicit library_items of the environment, and the only declarations that are directly visible are those that are explicit root library_items of the environment. Within a limited_with_clause, the only declarations that are visible are those that are the implicit declaration of the limited view of a library package of the environment, and the only declarations that are directly visible are those that are the implicit declaration of the limited view of a root library package.

Insert after paragraph 5:

Within a pragma that appears at the place of a compilation unit, the immediately preceding library_item and each of its ancestors is visible. The ancestor root library_item is directly visible.

the new paragraph:

Notwithstanding the rules of 4.1.3, an expanded name in a with_clause, a pragma in a context_clause, or a pragma that appears at the place of a compilation unit may consist of a prefix that denotes a generic package and a selector_name that denotes a child of that generic package. (The child is necessarily a generic unit; see 10.1.1.)

10.2 Program Execution

Replace paragraph 6:

- If a compilation unit with stubs is needed, then so are any corresponding subunits.

by:

- If a compilation unit with stubs is needed, then so are any corresponding subunits;
- If the (implicit) declaration of the limited view of a library package is needed, then so is the explicit declaration of the library package.

Replace paragraph 9:

The order of elaboration of library units is determined primarily by the *elaboration dependences*. There is an elaboration dependence of a given library_item upon another if the given library_item or any of its subunits depends semantically on the other library_item. In addition, if a given library_item or any of its subunits has a pragma Elaborate or Elaborate_All that mentions another library unit, then there is an elaboration dependence of the given library_item upon the body of the other library unit, and, for Elaborate_All only, upon each library_item needed by the declaration of the other library unit.

by:

The order of elaboration of library units is determined primarily by the *elaboration dependences*. There is an elaboration dependence of a given library_item upon another if the given library_item or any of its subunits depends semantically on the other library_item. In addition, if a given library_item or any of its subunits has a pragma Elaborate or Elaborate_All that names another library unit, then there is an elaboration dependence of the given library_item upon the body of the other library unit, and, for Elaborate_All only, upon each library_item needed by the declaration of the other library unit.

10.2.1 Elaboration Control

Insert after paragraph 4:

A pragma Preelaborate is a library unit pragma.

the new paragraphs:

The form of a pragma Preelaborable_Initialization is as follows:

pragma Preelaborable_Initialization(direct_name);

Replace paragraph 9:

- The creation of a default-initialized object (including a component) of a descendant of a private type, private extension, controlled type, task type, or protected type with entry_declarations; similarly the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

by:

- The creation of an object (including a component) of a type that does not have preelaborable initialization. Similarly, the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

Replace paragraph 10:

A generic body is preelaborable only if elaboration of a corresponding instance body would not perform any such actions, presuming that the actual for each formal private type (or extension) is a private type (or extension), and the actual for each formal subprogram is a user-defined subprogram.

by:

A generic body is preelaborable only if elaboration of a corresponding instance body would not perform any such actions, presuming that:

- the actual for each formal private type (or extension) declared within the formal part of the generic unit is a private type (or extension) that does not have preelaborable initialization;
- the actual for each formal type is nonstatic;
- the actual for each formal object is nonstatic; and
- the actual for each formal subprogram is a user-defined subprogram.

Insert after paragraph 11:

If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is *preelaborated*. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

the new paragraphs:

The following rules specify which entities have *preelaborable initialization*:

- The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, have preelaborable initialization if and only if the pragma Preelaborable_Initialization has been applied to them. A protected type with entry_declarations or a task type never has preelaborable initialization.
- A component (including a discriminant) of a record or protected type has preelaborable initialization if its declaration includes a default_expression whose execution does not perform any actions prohibited in preelaborable constructs as described above, or if its declaration does not include a default expression and its type has preelaborable initialization.
- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a derived record extension) if the non-inherited components all have preelaborable initialization. However, a user-defined controlled type with an overriding Initialize procedure does not have preelaborable initialization.
- A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, a record type whose components all have preelaborable initialization, or an interface type.

A pragma Preelaborable_Initialization specifies that a type has preelaborable initialization. This pragma shall appear in the visible part of a package or generic package.

If the pragma appears in the first list of basic_declarative_items of a package_specification, then the direct_name shall denote the first subtype of a private type, private extension, or protected type that is not an interface type and is without entry_declarations, and the type shall be declared immediately within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, each component of the protected type shall have preelaborable initialization. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit.

If the pragma appears in a generic_formal_part, then the direct_name shall denote a generic formal private type or a generic formal derived type declared in the same generic_formal_part as the pragma. In a generic_instantiation the corresponding actual type shall have preelaborable initialization.

Replace paragraph 16:

Legality Rules

A *pure* library_item is a preelaborable library_item that does not contain the declaration of any variable or named access type, except within a subprogram, generic subprogram, task unit, or protected unit.

by:

Static Semantics

A *pure* library_item is a preelaborable library_item whose elaboration does not perform any of the following actions:

- the elaboration of a variable declaration;
- the evaluation of an allocator of an access-to-variable type; for the purposes of this rule, the partial view of a type is presumed to have non-visible components whose default initialization evaluates such an allocator;
- the elaboration of the declaration of a named access-to-variable type unless the Storage_Size of the type has been specified by a static expression with value zero or is defined by the language to be zero;
- the elaboration of the declaration of a named access-to-constant type for which the Storage_Size has been specified by an expression other than a static expression with value zero.

The Storage_Size for an anonymous access-to-variable type declared at library level in a library unit that is declared pure is defined to be zero.

Legality Rules

Replace paragraph 17:

A pragma Pure is used to declare that a library unit is pure. If a pragma Pure applies to a library unit, then its compilation units shall be pure, and they shall depend semantically only on compilation units of other library units that are declared pure.

by:

A pragma Pure is used to declare that a library unit is pure. If a pragma Pure applies to a library unit, then its compilation units shall be pure, and they shall depend semantically only on compilation units of other library units that are declared pure. Furthermore, the full view of any partial view declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

Replace paragraph 18:

If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. Similarly, it may omit such a call and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters are of a limited type, and the addresses and values of all by-reference actual parameters, and the values of all by-copy-in actual parameters, are the same as they were at the earlier call. This permission applies even if the subprogram produces other side effects when called.

by:

If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters nor any object accessible via access values from the parameters are of a limited type, and the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call. This permission applies even if the subprogram produces other side effects when called.

Insert after paragraph 25:

If a pragma Elaborate_Body applies to a declaration, then the declaration requires a completion (a body).

the new paragraph:

The *library_unit_*name of a pragma Elaborate or Elaborate_All shall denote a nonlimited view of a library unit.

Section 11: Exceptions

11.3 Raise Statements

Replace paragraph 2:

raise_statement ::= **raise** [*exception*_name];

by:

raise_statement ::= **raise**; |
raise *exception*_name [**with** *string*_expression];

Insert after paragraph 3:

The name, if any, in a raise_statement shall denote an exception. A raise_statement with no *exception*_name (that is, a *re-raise statement*) shall be within a handler, but not within a body enclosed by that handler.

the new paragraph:

Name Resolution Rules

The expression, if any, in a raise_statement, is expected to be of type String.

Replace paragraph 4:

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a raise_statement with an *exception*_name, the named exception is raised. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

by:

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a raise_statement with an *exception*_name, the named exception is raised. If a *string*_expression is present, the expression is evaluated and its value is associated with the exception occurrence. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

Replace paragraph 6:

```
raise Ada.IO_Exceptions.Name_Error;    -- see A.13
```

by:

```
raise Ada.IO_Exceptions.Name_Error;    -- see A.13
raise Queue_Error with "Buffer Full"; -- see 9.11
```

11.4.1 The Package Exceptions

Replace paragraph 2:

```
package Ada.Exceptions is
    type Exception_Id is private;
    Null_Id : constant Exception_Id;
    function Exception_Name(Id : Exception_Id) return String;
```

by:

```
with Ada.Streams;
package Ada.Exceptions is
    pragma Preelaborate(Exceptions);
    type Exception_Id is private;
    pragma Preelaborable_Initialization(Exception_Id);
    Null_Id : constant Exception_Id;
    function Exception_Name(Id : Exception_Id) return String;
    function Wide_Exception_Name(Id : Exception_Id) return Wide_String;
    function Wide_Wide_Exception_Name(Id : Exception_Id)
```

```
        return Wide_Wide_String;
```

Replace paragraph 3:

```
        type Exception_Occurrence is limited private;
        type Exception_Occurrence_Access is access all Exception_Occurrence;
        Null_Occurrence : constant Exception_Occurrence;
```

by:

```
        type Exception_Occurrence is limited private;
        pragma Preelaborable_Initialization(Exception_Occurrence);
        type Exception_Occurrence_Access is access all Exception_Occurrence;
        Null_Occurrence : constant Exception_Occurrence;
```

Replace paragraph 4:

```
        procedure Raise_Exception(E : in Exception_Id;
                                  Message : in String := "");
        function Exception_Message(X : Exception_Occurrence) return String;
        procedure Reraise_Occurrence(X : in Exception_Occurrence);
```

by:

```
        procedure Raise_Exception(E : in Exception_Id;
                                  Message : in String := "");
            pragma No_Return(Raise_Exception);
        function Exception_Message(X : Exception_Occurrence) return String;
        procedure Reraise_Occurrence(X : in Exception_Occurrence);
```

Replace paragraph 5:

```
        function Exception_Identity(X : Exception_Occurrence)
                                    return Exception_Id;
        function Exception_Name(X : Exception_Occurrence) return String;
            -- Same as Exception_Name(Exception_Identity(X)).
        function Exception_Information(X : Exception_Occurrence) return String;
```

by:

```
        function Exception_Identity(X : Exception_Occurrence)
                                    return Exception_Id;
        function Exception_Name(X : Exception_Occurrence) return String;
            -- Same as Exception_Name(Exception_Identity(X)).
        function Wide_Exception_Name(X : Exception_Occurrence)
            return Wide_String;
            -- Same as Wide_Exception_Name(Exception_Identity(X)).
        function Wide_Wide_Exception_Name(X : Exception_Occurrence)
            return Wide_Wide_String;
            -- Same as Wide_Wide_Exception_Name(Exception_Identity(X)).
        function Exception_Information(X : Exception_Occurrence) return String;
```

Replace paragraph 6:

```
        procedure Save_Occurrence(Target : out Exception_Occurrence;
                                  Source : in Exception_Occurrence);
        function Save_Occurrence(Source : Exception_Occurrence)
                                 return Exception_Occurrence_Access;
    private
       ... -- not specified by the language
    end Ada.Exceptions;
```

by:

```
        procedure Save_Occurrence(Target : out Exception_Occurrence;
                                  Source : in Exception_Occurrence);
        function Save_Occurrence(Source : Exception_Occurrence)
                                 return Exception_Occurrence_Access;

        procedure Read_Exception_Occurrence
```

```
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : out Exception_Occurrence);
   procedure Write_Exception_Occurrence
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : in Exception_Occurrence);

   for Exception_Occurrence'Read use Read_Exception_Occurrence;
   for Exception_Occurrence'Write use Write_Exception_Occurrence;

private
   ... -- not specified by the language
end Ada.Exceptions;
```

Replace paragraph 10:

Raise_Exception raises a new occurrence of the identified exception. In this case, Exception_Message returns the Message parameter of Raise_Exception. For a raise_statement with an *exception*_name, Exception_Message returns implementation-defined information about the exception occurrence. Reraise_Occurrence reraises the specified exception occurrence.

by:

Raise_Exception raises a new occurrence of the identified exception.

Exception_Message returns the message associated with the given Exception_Occurrence. For an occurrence raised by a call to Raise_Exception, the message is the Message parameter passed to Raise_Exception. For the occurrence raised by a raise_statement with an *exception*_name and a *string*_expression, the message is the *string*_expression. For the occurrence raised by a raise_statement with an *exception*_name but without a *string*_expression, the message is a string giving implementation-defined information about the exception occurrence. In all cases, Exception_Message returns a string with lower bound 1.

Reraise_Occurrence reraises the specified exception occurrence.

Replace paragraph 12:

The Exception_Name functions return the full expanded name of the exception, in upper case, starting with a root library unit. For an exception declared immediately within package Standard, the defining_identifier is returned. The result is implementation defined if the exception is declared within an unnamed block_statement.

by:

The Wide_Wide_Exception_Name functions return the full expanded name of the exception, in upper case, starting with a root library unit. For an exception declared immediately within package Standard, the defining_identifier is returned. The result is implementation defined if the exception is declared within an unnamed block_statement.

The Exception_Name functions (respectively, Wide_Exception_Name) return the same sequence of graphic characters as that defined for Wide_Wide_Exception_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Exception_Name for the same value of the argument.

The string returned by the Exception_Name, Wide_Exception_Name, and Wide_Wide_Exception_Name functions has lower bound 1.

Replace paragraph 13:

Exception_Information returns implementation-defined information about the exception occurrence.

by:

Exception_Information returns implementation-defined information about the exception occurrence. The returned string has lower bound 1.

Replace paragraph 14:

Raise_Exception and Reraise_Occurrence have no effect in the case of Null_Id or Null_Occurrence. Exception_Message, Exception_Identity, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Id or Null_Occurrence.

by:

Reraise_Occurrence has no effect in the case of Null_Occurrence. Raise_Exception and Exception_Name raise Constraint_Error for a Null_Id. Exception_Message, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence. Exception_Identity applied to Null_Occurrence returns Null_Id.

Replace paragraph 16:

Implementation Requirements

The implementation of the Write attribute (see 13.13.2) of Exception_Occurrence shall support writing a representation of an exception occurrence to a stream; the implementation of the Read attribute of Exception_Occurrence shall support reconstructing an exception occurrence from a stream (including one written in a different partition).

by:

Write_Exception_Occurrence writes a representation of an exception occurrence to a stream; Read_Exception_Occurrence reconstructs an exception occurrence from a stream (including one written in a different partition).

11.4.2 Pragmas Assert and Assertion_Policy

Insert new clause:

Pragma Assert is used to assert the truth of a Boolean expression at any point within a sequence of declarations or statements. Pragma Assertion_Policy is used to control whether such assertions are to be ignored by the implementation, checked at run-time, or handled in some implementation-defined manner.

Syntax

The form of a pragma Assert is as follows:

pragma Assert([Check =>] *boolean_*expression[, [Message =>] *string_*expression]);

A pragma Assert is allowed at the place where a declarative_item or a statement is allowed.

The form of a pragma Assertion_Policy is as follows:

pragma Assertion_Policy(*policy_*identifier);

A pragma Assertion_Policy is a configuration pragma.

Name Resolution Rules

The expected type for the *boolean_*expression of a pragma Assert is any boolean type. The expected type for the *string_*expression of a pragma Assert is type String.

Legality Rules

The *policy_*identifier of a pragma Assertion_Policy shall be either Check, Ignore, or an implementation-defined identifier.

Static Semantics

A pragma Assertion_Policy is a configuration pragma that specifies the assertion policy in effect for the compilation units to which it applies. Different policies may apply to different compilation units within the same partition. The default assertion policy is implementation-defined.

The following language-defined library package exists:

```
package Ada.Assertions is
    pragma Pure(Assertions);

    Assertion_Error : exception;

    procedure Assert(Check : in Boolean);
    procedure Assert(Check : in Boolean; Message : in String);

end Ada.Assertions;
```

A compilation unit containing a pragma Assert has a semantic dependence on the Assertions library unit.

The assertion policy that applies to a generic unit also applies to all its instances.

Dynamic Semantics

An assertion policy specifies how a pragma Assert is interpreted by the implementation. If the assertion policy is Ignore at the point of a pragma Assert, the pragma is ignored. If the assertion policy is Check at the point of a pragma Assert, the elaboration of the pragma consists of evaluating the boolean expression, and if the result is False, evaluating the Message argument, if any, and raising the exception Assertions.Assertion_Error, with a message if the Message argument is provided.

Calling the procedure Assertions.Assert without a Message parameter is equivalent to:

```
if Check = False then
   raise Ada.Assertions.Assertion_Error;
end if;
```

Calling the procedure Assertions.Assert with a Message parameter is equivalent to:

```
if Check = False then
   raise Ada.Assertions.Assertion_Error with Message;
end if;
```

The procedures Assertions.Assert have these effects independently of the assertion policy in effect.

Implementation Permissions

Assertion_Error may be declared by renaming an implementation-defined exception from another package.

Implementations may define their own assertion policies.

NOTES

2 Normally, the boolean expression in a pragma Assert should not call functions that have significant side-effects when the result of the expression is True, so that the particular assertion policy in effect will not affect normal operation of the program.

11.4.3 Example of Exception Handling

Replace paragraph 2:

```
with Ada.Exceptions;
use Ada;
package File_System is
    type File_Handle is limited private;
```

by:

```
package File_System is
    type File_Handle is limited private;
```

Replace paragraph 6:

```
package body File_System is
    procedure Open(F : in out File_Handle; Name : String) is
    begin
        if File_Exists(Name) then
            ...
        else
            Exceptions.Raise_Exception(File_Not_Found'Identity,
                                       "File not found: " & Name & ".");
        end if;
    end Open;
```

by:

```
package body File_System is
    procedure Open(F : in out File_Handle; Name : String) is
    begin
```

```
        if File_Exists(Name) then
            ...
        else
            raise File_Not_Found with "File not found: " & Name & ".";
        end if;
    end Open;
```

11.5 Suppressing Checks

Replace paragraph 1:

A pragma Suppress gives permission to an implementation to omit certain language-defined checks.

by:

Checking pragmas give instructions to an implementation on handling language-defined checks. A pragma Suppress gives permission to an implementation to omit certain language-defined checks, while a pragma Unsuppress revokes the permission to omit checks.

Replace paragraph 3:

The form of a pragma Suppress is as follows:

by:

The forms of checking pragmas are as follows:

Replace paragraph 4:

pragma Suppress(identifier [, [On =>] name]);

by:

pragma Suppress(identifier);

pragma Unsuppress(identifier);

Replace paragraph 5:

A pragma Suppress is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.

by:

A checking pragma is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.

Replace paragraph 6:

The identifier shall be the name of a check. The name (if present) shall statically denote some entity.

by:

The identifier shall be the name of a check.

Delete paragraph 7:

For a pragma Suppress that is immediately within a package_specification and includes a name, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification.

Replace paragraph 8:

A pragma Suppress gives permission to an implementation to omit the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification and includes a name, to the end of the scope of the named entity. If the pragma includes a name, the permission applies only to checks performed on the named entity, or, for a subtype, on objects and values of its type. Otherwise, the permission applies to all entities. If permission has been given to suppress a given check, the check is said to be *suppressed*.

by:

A checking pragma applies to the named check in a specific region, and applies to all entities in that region. A checking pragma given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a checking pragma given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies.

If a checking pragma applies to a generic instantiation, then the checking pragma also applies to the instance. If a checking pragma applies to a call to a subprogram that has a pragma Inline applied to it, then the checking pragma also applies to the inlined subprogram body.

A pragma Suppress gives permission to an implementation to omit the named check (or every check in the case of All_Checks) for any entities to which it applies. If permission has been given to suppress a given check, the check is said to be *suppressed*.

A pragma Unsuppress revokes the permission to omit the named check (or every check in the case of All_Checks) given by any pragma Suppress that applies at the point of the pragma Unsuppress. The permission is revoked for the region to which the pragma Unsuppress applies. If there is no such permission at the point of a pragma Unsuppress, then the pragma has no effect. A later pragma Suppress can renew the permission.

Replace paragraph 11:

When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When passing an actual parameter to a formal access parameter, check that the value of the actual parameter is not **null**. When evaluating a discriminant_association for an access discriminant, check that the value of the discriminant is not **null**.

by:

When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When converting to a subtype that excludes null, check that the converted value is not **null**.

Replace paragraph 13:

Division_Check
: Check that the second operand is not zero for the operations /, rem and mod.

by:

Division_Check
: Check that the second operand is not zero for the operations /, **rem** and **mod**.

Insert before paragraph 20:

Elaboration_Check
: When a subprogram or protected entry 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.

the new paragraphs:

Accessibility_Check
: Check the accessibility level of an entity or view.

Allocation_Check
: For an allocator, check that the master of any tasks to be created by the allocator is not yet completed or some dependents have not yet terminated, and that the finalization of the collection has not started.

Delete paragraph 21:

Accessibility_Check
: Check the accessibility level of an entity or view.

Replace paragraph 27:

An implementation is allowed to place restrictions on Suppress pragmas. An implementation is allowed to add additional check names, with implementation-defined semantics. When Overflow_Check has been suppressed, an implementation may also suppress an unspecified subset of the Range_Checks.

by:

An implementation is allowed to place restrictions on checking pragmas, subject only to the requirement that pragma Unsuppress shall allow any check names supported by pragma Suppress. An implementation is allowed to add additional check names, with implementation-defined semantics. When Overflow_Check has been suppressed, an implementation may also suppress an unspecified subset of the Range_Checks.

An implementation may support an additional parameter on pragma Unsuppress similar to the one allowed for pragma Suppress (see J.10). The meaning of such a parameter is implementation-defined.

Insert after paragraph 29:

2 There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons.

the new paragraph:

3 It is possible to give both a pragma Suppress and Unsuppress for the same check immediately within the same declarative_part. In that case, the last pragma given determines whether or not the check is suppressed. Similarly, it is possible to resuppress a check which has been unsuppressed by giving a pragma Suppress in an inner declarative region.

Replace paragraph 30:

Examples of suppressing checks:

by:

Examples of suppressing and unsuppressing checks:

Replace paragraph 32:

```
pragma Suppress(Range_Check);
pragma Suppress(Index_Check, On => Table);
```

by:

```
pragma Suppress(Index_Check);
pragma Unsuppress(Overflow_Check);
```

Section 12: Generic Units

12.1 Generic Declarations

Replace paragraph 8:

A generic_declaration declares a generic unit — a generic package, generic procedure or generic function, as appropriate.

by:

A generic_declaration declares a generic unit — a generic package, generic procedure, or generic function, as appropriate.

12.3 Generic Instantiation

Replace paragraph 2:

```
generic_instantiation ::=
    package defining_program_unit_name is
      new generic_package_name [generic_actual_part];
  | procedure defining_program_unit_name is
      new generic_procedure_name [generic_actual_part];
  | function defining_designator is
      new generic_function_name [generic_actual_part];
```

by:

```
generic_instantiation ::=
    package defining_program_unit_name is
      new generic_package_name [generic_actual_part];
  | [overriding_indicator]
    procedure defining_program_unit_name is
      new generic_procedure_name [generic_actual_part];
  | [overriding_indicator]
    function defining_designator is
      new generic_function_name [generic_actual_part];
```

12.4 Formal Objects

Replace paragraph 2:

```
formal_object_declaration ::=
   defining_identifier_list : mode subtype_mark [:= default_expression]
```

by:

```
formal_object_declaration ::=
   defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : mode access_definition [:= default_expression];
```

Replace paragraph 5:

For a generic formal object of mode **in out**, the type of the actual shall resolve to the type of the formal.

by:

For a generic formal object of mode **in out**, the type of the actual shall resolve to the type determined by the subtype_mark, or for a formal_object_declaration with an access_definition, to a specific anonymous access type. If the anonymous access type is an access-to-object type, the type of the actual shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the actual shall have a designated profile which is type conformant with that of the access_definition.

Insert after paragraph 7:

For a generic formal object of mode **in**, the actual shall be an expression. For a generic formal object of mode **in out**, the actual shall be a name that denotes a variable for which renaming is allowed (see 8.5.1).

the new paragraphs:

In the case where the type of the formal is defined by an access_definition, the type of the actual and the type of the formal:

- shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or
- shall both be access-to-subprogram types with subtype conformant designated profiles.

For a formal_object_declaration with a null_exclusion or an access_definition that has a null_exclusion:

- if the actual matching the formal_object_declaration denotes the generic formal object of another generic unit *G*, and the instantiation containing the actual occurs within the body of *G* or within the body of a generic unit declared within the declarative region of *G*, then the declaration of the formal object of *G* shall have a null_exclusion;
- otherwise, the subtype of the actual matching the formal_object_declaration shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Delete paragraph 8:

The type of a generic formal object of mode **in** shall be nonlimited.

Replace paragraph 9:

A formal_object_declaration declares a generic formal object. The default mode is **in**. For a formal object of mode **in**, the nominal subtype is the one denoted by the subtype_mark in the declaration of the formal. For a formal object of mode **in out**, its type is determined by the subtype_mark in the declaration; its nominal subtype is nonstatic, even if the subtype_mark denotes a static subtype.

by:

A formal_object_declaration declares a generic formal object. The default mode is **in**. For a formal object of mode **in**, the nominal subtype is the one denoted by the subtype_mark or access_definition in the declaration of the formal. For a formal object of mode **in out**, its type is determined by the subtype_mark or access_definition in the declaration; its nominal subtype is nonstatic, even if the subtype_mark denotes a static subtype; for a composite type, its nominal subtype is unconstrained if the first subtype of the type is unconstrained, even if the subtype_mark denotes a constrained subtype.

Replace paragraph 10:

In an instance, a formal_object_declaration of mode **in** declares a new stand-alone constant object whose initialization expression is the actual, whereas a formal_object_declaration of mode **in out** declares a view whose properties are identical to those of the actual.

by:

In an instance, a formal_object_declaration of mode **in** is a *full constant declaration* and declares a new stand-alone constant object whose initialization expression is the actual, whereas a formal_object_declaration of mode **in out** declares a view whose properties are identical to those of the actual.

12.5 Formal Types

Replace paragraph 1:

A generic formal subtype can be used to pass to a generic unit a subtype whose type is in a certain class of types.

by:

A generic formal subtype can be used to pass to a generic unit a subtype whose type is in a certain category of types.

Replace paragraph 3:

```
formal_type_definition ::=
     formal_private_type_definition
   | formal_derived_type_definition
   | formal_discrete_type_definition
   | formal_signed_integer_type_definition
   | formal_modular_type_definition
   | formal_floating_point_definition
   | formal_ordinary_fixed_point_definition
   | formal_decimal_fixed_point_definition
   | formal_array_type_definition
   | formal_access_type_definition
```

by:

```
formal_type_definition ::=
     formal_private_type_definition
   | formal_derived_type_definition
   | formal_discrete_type_definition
   | formal_signed_integer_type_definition
   | formal_modular_type_definition
   | formal_floating_point_definition
   | formal_ordinary_fixed_point_definition
   | formal_decimal_fixed_point_definition
   | formal_array_type_definition
   | formal_access_type_definition
   | formal_interface_type_definition
```

Replace paragraph 6:

The form of a formal_type_definition *determines a class* to which the formal type belongs. For a formal_private_type_definition the reserved words **tagged** and **limited** indicate the class (see 12.5.1). For a formal_derived_type_definition the class is the derivation class rooted at the ancestor type. For other formal types, the name of the syntactic category indicates the class; a formal_discrete_type_definition defines a discrete type, and so on.

by:

The form of a formal_type_definition *determines a category (of types)* to which the formal type belongs. For a formal_private_type_definition the reserved words **tagged** and **limited** indicate the category of types (see 12.5.1). For a formal_derived_type_definition the category of types is the derivation class rooted at the ancestor type. For other formal types, the name of the syntactic category indicates the category of types; a formal_discrete_type_definition defines a discrete type, and so on.

Replace paragraph 7:

The actual type shall be in the class determined for the formal.

by:

The actual type shall be in the category determined for the formal.

Replace paragraph 8:

The formal type also belongs to each class that contains the determined class. The primitive subprograms of the type are as for any type in the determined class. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later in its immediate scope according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

by:

The formal type also belongs to each category that contains the determined category. The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

12.5.1 Formal Private and Derived Types

Replace paragraph 1:

The class determined for a formal private type can be either limited or nonlimited, and either tagged or untagged; no more specific class is known for such a type. The class determined for a formal derived type is the derivation class rooted at the ancestor type.

by:

In its most general form, the category determined for a formal private type is all types, but it can be restricted to only nonlimited types or to only tagged types. The category determined for a formal derived type is the derivation class rooted at the ancestor type.

Replace paragraph 3:

formal_derived_type_definition ::= [**abstract**] **new** subtype_mark [**with private**]

by:

formal_derived_type_definition ::=
[**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list] **with private**]

Replace paragraph 5:

The *ancestor subtype* of a formal derived type is the subtype denoted by the subtype_mark of the formal_derived_type_definition. For a formal derived type declaration, the reserved words **with private** shall appear if and only if the ancestor type is a tagged type; in this case the formal derived type is a private extension of the ancestor type and the ancestor shall not be a class-wide type. Similarly, the optional reserved word **abstract** shall appear only if the ancestor type is a tagged type.

by:

The *ancestor subtype* of a formal derived type is the subtype denoted by the subtype_mark of the formal_derived_type_definition. For a formal derived type declaration, the reserved words **with private** shall appear if and only if the ancestor type is a tagged type; in this case the formal derived type is a private extension of the ancestor type and the ancestor shall not be a class-wide type. Similarly, an interface_list or the optional reserved words **abstract** or **synchronized** shall appear only if the ancestor type is a tagged type. The reserved word **limited** or **synchronized** shall appear only if the ancestor type and any progenitor types are limited types. The reserved word **synchronized** shall appear (rather than **limited**) if the ancestor type or any of the progenitor types are synchronized interfaces.

The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

Insert after paragraph 10:

- If the ancestor subtype is an unconstrained discriminated subtype, then the actual shall have the same number of discriminants, and each discriminant of the actual shall correspond to a discriminant of the ancestor, in the sense of 3.7.

the new paragraph:

- If the ancestor subtype is an access subtype, the actual subtype shall exclude null if and only if the ancestor subtype excludes null.

Replace paragraph 16:

The class determined for a formal private type is as follows:

by:

The category determined for a formal private type is as follows:

Replace paragraph 17:

Type Definition	*Determined Class*
limited private	the class of all types
private	the class of all nonlimited types
tagged limited private	the class of tagged types
tagged private	the class of all nonlimited tagged types

by:

Type Definition	*Determined Category*
limited private	the category of all types
private	the category of all nonlimited types
tagged limited private	the category of all tagged types
tagged private	the category of all nonlimited tagged types

Replace paragraph 20:

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new discriminant_part is not specified), as for a derived type defined by a derived_type_definition (see 3.4).

by:

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new discriminant_part is not specified), as for a derived type defined by a derived_type_definition (see 3.4 and 7.3.1).

Replace paragraph 21:

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type, and are implicitly declared at the earliest place, if any, within the immediate scope of the formal type, where the corresponding primitive subprogram of the ancestor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

by:

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type and any progenitor types, and are implicitly declared at the earliest place, if any, immediately within the declarative region in which the formal type is declared, where the corresponding primitive subprogram of the ancestor or progenitor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the

corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

Insert after paragraph 23:

S'Definite

S'Definite yields True if the actual subtype corresponding to S is definite; otherwise it yields False. The value of this attribute is of the predefined type Boolean.

the new paragraphs:

Dynamic Semantics

In the case where a formal type is tagged with unknown discriminants, and the actual type is a class-wide type *T*'Class:

- For the purposes of defining the primitive operations of the formal type, each of the primitive operations of the actual type is considered to be a subprogram (with an intrinsic calling convention — see 6.3.1) whose body consists of a dispatching call upon the corresponding operation of *T*, with its formal parameters as the actual parameters. If it is a function, the result of the dispatching call is returned.
- If the corresponding operation of *T* has no controlling formal parameters, then the controlling tag value is determined by the context of the call, according to the rules for tag-indeterminate calls (see 3.9.2 and 5.2). In the case where the tag would be statically determined to be that of the formal type, the call raises Program_Error. If such a function is renamed, any call on the renaming raises Program_Error.

Replace paragraph 24:

9 In accordance with the general rule that the actual type shall belong to the class determined for the formal (see 12.5, "Formal Types"):

by:

9 In accordance with the general rule that the actual type shall belong to the category determined for the formal (see 12.5, "Formal Types"):

12.5.2 Formal Scalar Types

Replace paragraph 1:

A *formal scalar type* is one defined by any of the formal_type_definitions in this subclause. The class determined for a formal scalar type is discrete, signed integer, modular, floating point, ordinary fixed point, or decimal.

by:

A *formal scalar type* is one defined by any of the formal_type_definitions in this subclause. The category determined for a formal scalar type is the category of all discrete, signed integer, modular, floating point, ordinary fixed point, or decimal types.

12.5.3 Formal Array Types

Replace paragraph 1:

The class determined for a formal array type is the class of all array types.

by:

The category determined for a formal array type is the category of all array types.

12.5.4 Formal Access Types

Replace paragraph 1:

The class determined for a formal access type is the class of all access types.

by:

The category determined for a formal access type is the category of all access types.

Replace paragraph 4:

If and only if the general_access_modifier **constant** applies to the formal, the actual shall be an access-to-constant type. If the general_access_modifier **all** applies to the formal, then the actual shall be a general access-to-variable type (see 3.10).

by:

If and only if the general_access_modifier **constant** applies to the formal, the actual shall be an access-to-constant type. If the general_access_modifier **all** applies to the formal, then the actual shall be a general access-to-variable type (see 3.10). If and only if the formal subtype excludes null, the actual subtype shall exclude null.

12.5.5 Formal Interface Types

Insert new clause:

The category determined for a formal interface type is the category of all interface types.

Syntax

formal_interface_type_definition ::= interface_type_definition

Legality Rules

The actual type shall be a descendant of every progenitor of the formal type.

The actual type shall be a limited, task, protected, or synchronized interface if and only if the formal type is also, respectively, a limited, task, protected, or synchronized interface.

Examples

```
type Root_Work_Item is tagged private;

generic
   type Managed_Task is task interface;
   type Work_Item(<>) is new Root_Work_Item with private;
package Server_Manager is
   task type Server is new Managed_Task with
      entry Start(Data : in out Work_Item);
   end Server;
end Server_Manager;
```

This generic allows an application to establish a standard interface that all tasks need to implement so they can be managed appropriately by an application-specific scheduler.

12.6 Formal Subprograms

Replace paragraph 2:

formal_subprogram_declaration ::= **with** subprogram_specification [**is** subprogram_default];

by:

formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
  | formal_abstract_subprogram_declaration
formal_concrete_subprogram_declaration ::=
   with subprogram_specification [**is** subprogram_default];
formal_abstract_subprogram_declaration ::=

with subprogram_specification **is abstract** [subprogram_default];

Replace paragraph 3:

subprogram_default ::= default_name | <>

by:

subprogram_default ::= default_name | <> | **null**

Insert after paragraph 4:

default_name ::= name

the new paragraph:

A subprogram_default of **null** shall not be specified for a formal function or for a formal_abstract_subprogram_declaration.

Insert after paragraph 8:

The profiles of the formal and actual shall be mode-conformant.

the new paragraphs:

For a parameter or result subtype of a formal_subprogram_declaration that has an explicit null_exclusion:

- if the actual matching the formal_subprogram_declaration denotes a generic formal object of another generic unit *G*, and the instantiation containing the actual that occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result type of the formal subprogram of *G* shall have a null_exclusion;
- otherwise, the subtype of the corresponding parameter or result type of the actual matching the formal_subprogram_declaration shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

If a formal parameter of a formal_abstract_subprogram_declaration is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a *controlling type* of the formal_abstract_subprogram_declaration. Similarly, if the result of a formal_abstract_subprogram_declaration for a function is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a controlling type of the formal_abstract_subprogram_declaration. A formal_abstract_subprogram_declaration shall have exactly one controlling type.

The actual subprogram for a formal_abstract_subprogram_declaration shall be a dispatching operation of the controlling type or of the actual type corresponding to the controlling type.

Insert after paragraph 10:

If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal.

the new paragraphs:

If a generic unit has a subprogram_default specified by the reserved word **null**, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a null procedure having the profile given in the formal_subprogram_declaration.

The subprogram declared by a formal_abstract_subprogram_declaration with a controlling type *T* is a dispatching operation of type *T*.

Replace paragraph 16:

18 The actual subprogram cannot be abstract (see 3.9.3).

by:

18 The actual subprogram cannot be abstract unless the formal subprogram is a formal_abstract_subprogram_declaration (see 3.9.3).

19 The subprogram declared by a formal_abstract_subprogram_declaration is an abstract subprogram. All calls on a subprogram declared by a formal_abstract_subprogram_declaration must be dispatching calls. See 3.9.3.

20 A null procedure as a subprogram default has convention Intrinsic (see 6.3.1).

Replace paragraph 18:

```
with function "+"(X, Y : Item) return Item is <>;
with function Image(X : Enum) return String is Enum'Image;
with procedure Update is Default_Update;
```

by:

```
with function "+"(X, Y : Item) return Item is <>;
with function Image(X : Enum) return String is Enum'Image;
with procedure Update is Default_Update;
with procedure Pre_Action(X : in Item) is null;  -- defaults to no action
with procedure Write(S    : not null access Root_Stream_Type'Class;
                     Desc : Descriptor)
                     is abstract Descriptor'Write;  -- see 13.13.2
-- Dispatching operation on Descriptor with default
```

12.7 Formal Packages

Replace paragraph 3:

formal_package_actual_part ::=
(<>) | [generic_actual_part]

by:

formal_package_actual_part ::=
([**others** =>] <>)
| [generic_actual_part]
| (formal_package_association {, formal_package_association}, **others** => <>)

formal_package_association ::=
generic_association
| *generic_formal_parameter_*selector_name => <>

Any positional formal_package_associations shall precede any named formal_package_associations.

Insert after paragraph 4:

The *generic_package_*name shall denote a generic package (the *template* for the formal package); the formal package is an instance of the template.

the new paragraph:

A formal_package_actual_part shall contain at most one formal_package_association for each formal parameter. If the formal_package_actual_part does not include "**others** => <>", each formal parameter without an association shall have a default_expression or subprogram_default.

Replace paragraph 5:

The actual shall be an instance of the template. If the formal_package_actual_part is (<>), then the actual may be any instance of the template; otherwise, each actual parameter of the actual instance shall match the corresponding actual parameter of the formal package (whether the actual parameter is given explicitly or by default), as follows:

by:

The actual shall be an instance of the template. If the formal_package_actual_part is (<>) or (**others** => <>), then the actual may be any instance of the template; otherwise, certain of the actual parameters of the actual instance shall match the corresponding actual parameters of the formal package, determined as follows:

- If the formal_package_actual_part includes generic_associations as well as associations with <>, then only the actual parameters specified explicitly with generic_associations are required to match;

- Otherwise, all actual parameters shall match, whether any actual parameter is given explicitly or by default.

The rules for matching of actual parameters between the actual instance and the formal package are as follows:

Replace paragraph 6:

- For a formal object of mode **in** the actuals match if they are static expressions with the same value, or if they statically denote the same constant, or if they are both the literal **null**.

by:

- For a formal object of mode **in**, the actuals match if they are static expressions with the same value, or if they statically denote the same constant, or if they are both the literal **null**.

Replace paragraph 10:

The visible part of a formal package includes the first list of basic_declarative_items of the package_specification. In addition, if the formal_package_actual_part is (<>), it also includes the generic_formal_part of the template for the formal package.

by:

The visible part of a formal package includes the first list of basic_declarative_items of the package_specification. In addition, for each actual parameter that is not required to match, a copy of the declaration of the corresponding formal parameter of the template is included in the visible part of the formal package. If the copied declaration is for a formal type, copies of the implicit declarations of the primitive subprograms of the formal type are also included in the visible part of the formal package.

For the purposes of matching, if the actual instance *A* is itself a formal package, then the actual parameters of *A* are those specified explicitly or implicitly in the formal_package_actual_part for *A*, plus, for those not specified, the copies of the formal parameters of the template included in the visible part of *A*.

Insert after paragraph 11:

For the purposes of matching, if the actual instance *A* is itself a formal package, then the actual parameters of *A* are those specified explicitly or implicitly in the formal_package_actual_part for *A*, plus, for those not specified, the copies of the formal parameters of the template included in the visible part of *A*.

the new paragraphs:

Examples

Example of a generic package with formal package parameters:

```
with Ada.Containers.Ordered_Maps;  -- see A.18.6
generic
   with package Mapping_1 is new Ada.Containers.Ordered_Maps(<>);
   with package Mapping_2 is new Ada.Containers.Ordered_Maps
                                    (Key_Type => Mapping_1.Element_Type,
                                     others => <>);
package Ordered_Join is
   -- Provide a "join" between two mappings

   subtype Key_Type is Mapping_1.Key_Type;
   subtype Element_Type is Mapping_2.Element_Type;

   function Lookup(Key : Key_Type) return Element_Type;

   ...
end Ordered_Join;
```

Example of an instantiation of a package with formal packages:

```
with Ada.Containers.Ordered_Maps;
package Symbol_Package is

   type String_Id is ...
```

```
        type Symbol_Info is ...

        package String_Table is new Ada.Containers.Ordered_Maps
                (Key_Type => String,
                 Element_Type => String_Id);

        package Symbol_Table is new Ada.Containers.Ordered_Maps
                (Key_Type => String_Id,
                 Element_Type => Symbol_Info);

        package String_Info is new Ordered_Join(Mapping_1 => String_Table,
                                                Mapping_2 => Symbol_Table);

        Apple_Info : constant Symbol_Info := String_Info.Lookup("Apple");
```

end Symbol_Package;

Section 13: Representation Issues

13.1 Representation Items

Replace paragraph 7:

The *representation* of an object consists of a certain number of bits (the *size* of the object). These are the bits that are normally read or updated by the machine code when loading, storing, or operating-on the value of the object. This includes some padding bits, when the size of the object is greater than the size of its subtype. Such padding bits are considered to be part of the representation of the object, rather than being gaps between objects, if these bits are normally read and updated.

by:

The *representation* of an object consists of a certain number of bits (the *size* of the object). For an object of an elementary type, these are the bits that are normally read or updated by the machine code when loading, storing, or operating-on the value of the object. For an object of a composite type, these are the bits reserved for this object, and include bits occupied by subcomponents of the object. If the size of an object is greater than that of its subtype, the additional bits are padding bits. For an elementary object, these padding bits are normally read and updated along with the others. For a composite object, padding bits might not be read or updated in any given composite operation, depending on the implementation.

Replace paragraph 11:

Operational and representation aspects of a generic formal parameter are the same as those of the actual. Operational and representation aspects of a partial view are the same as those of the full view. A type-related representation item is not allowed for a descendant of a generic formal untagged type.

by:

Operational and representation aspects of a generic formal parameter are the same as those of the actual. Operational and representation aspects are the same for all views of a type. A type-related representation item is not allowed for a descendant of a generic formal untagged type.

Insert after paragraph 13:

A representation or operational item that is not supported by the implementation is illegal, or raises an exception at run time.

the new paragraph:

A type_declaration is illegal if it has one or more progenitors, and a representation item applies to an ancestor, and this representation item conflicts with the representation of some other ancestor. The cases that cause conflicts are implementation defined.

Replace paragraph 15.1:

In contrast, whether operational aspects are inherited by a derived type depends on each specific aspect. When operational aspects are inherited by a derived type, aspects that were directly specified before the declaration of the derived type, or (in the case where the parent is derived) that were inherited by the parent type from the grandparent type are inherited. An inherited operational aspect is overridden by a subsequent operational item that specifies the same aspect of the type.

by:

In contrast, whether operational aspects are inherited by an untagged derived type depends on each specific aspect. Operational aspects are never inherited for a tagged type. When operational aspects are inherited by an untagged derived type, aspects that were directly specified by operational items that are visible at the point of the derived type declaration, or (in the case where the parent is derived) that were inherited by the parent type from the grandparent type are inherited. An inherited operational aspect is overridden by a subsequent operational item that specifies the same aspect of the type.

When an aspect that is a subprogram is inherited, the derived type inherits the aspect in the same way that a derived type inherits a user-defined primitive subprogram from its parent (see 3.4).

Insert after paragraph 18.1:

If an operational aspect is *specified* for an entity (meaning that it is either directly specified or inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value for that aspect.

the new paragraph:

A representation item that specifies an aspect of representation that would have been chosen in the absence of the representation item is said to be *confirming*.

Insert after paragraph 21:

The recommended level of support for all representation items is qualified as follows:

the new paragraph:

- A confirming representation item should be supported.

Replace paragraph 24:

- An aliased component, or a component whose type is by-reference, should always be allocated at an addressable location.

by:

- An implementation need not support a nonconfirming representation item if it could cause an aliased object or an object of a by-reference type to be allocated at a nonaddressable location or, when the alignment attribute of the subtype of such an object is nonzero, at an address that is not an integral multiple of that alignment.
- An implementation need not support a nonconfirming representation item if it could cause an aliased object of an elementary type to have a size other than that which would have been chosen by default.
- An implementation need not support a nonconfirming representation item if it could cause an aliased object of a composite type, or an object whose type is by-reference, to have a size smaller than that which would have been chosen by default.
- An implementation need not support a nonconfirming subtype-specific representation item specifying an aspect of representation of an indefinite or abstract subtype.

For purposes of these rules, the determination of whether a representation item applied to a type *could cause* an object to have some property is based solely on the properties of the type itself, not on any available information about how the type is used. In particular, it presumes that minimally aligned objects of this type might be declared at some point.

13.2 Pragma Pack

Insert after paragraph 6:

If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.

the new paragraph:

If a packed type has a component that is not of a by-reference type and has no aliased part, then such a component need not be aligned according to the Alignment of its subtype; in particular it need not be allocated on a storage element boundary.

13.3 Representation Attributes

Insert after paragraph 8:

A *storage element* is an addressable element of storage in the machine. A *word* is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of an integral number of storage elements.

the new paragraph:

A *machine scalar* is an amount of storage that can be conveniently and efficiently loaded, stored, or operated upon by the hardware. Machine scalars consist of an integral number of storage elements. The set of machine scalars is implementation defined, but must include at least the storage element and the word. Machine scalars are used to interpret component_clauses when the nondefault bit ordering applies.

Delete paragraph 18:

- Objects (including subcomponents) that are aliased or of a by-reference type should be allocated on storage element boundaries.

Replace paragraph 22:

For a prefix X that denotes a subtype or object:

by:

For a prefix X that denotes an object:

Replace paragraph 23:

X'Alignment

The Address of an object that is allocated under control of the implementation is an integral multiple of the Alignment of the object (that is, the Address modulo the Alignment is zero). The offset of a record component is a multiple of the Alignment of the component. For an object that is not allocated under control of the implementation (that is, one that is imported, that is allocated by a user-defined allocator, whose Address has been specified, or is designated by an access value returned by an instance of Unchecked_Conversion), the implementation may assume that the Address is an integral multiple of its Alignment. The implementation shall not assume a stricter alignment.

by:

X'Alignment

The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero).

Delete paragraph 24:

The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary.

Replace paragraph 25:

Alignment may be specified for first subtypes and stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. If the Alignment of a subtype is specified, then the Alignment of an object of the subtype is at least as strict, unless the object's Alignment is also specified. The Alignment of an object created by an allocator is that of the designated subtype.

by:

Alignment may be specified for stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative.

For every subtype S:

S'Alignment

The value of this attribute is of type *universal_integer*, and nonnegative.

For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item.

Alignment may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative.

Delete paragraph 26:

If an Alignment is specified for a composite subtype or object, this Alignment shall be equal to the least common multiple of any specified Alignments of the subcomponent subtypes, or an integer multiple thereof.

Replace paragraph 28:

If the Alignment is specified for an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to the Alignment.

by:

For an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to its Alignment.

Replace paragraph 30:

- An implementation should support specified Alignments that are factors and multiples of the number of storage elements per word, subject to the following:

by:

- An implementation should support an Alignment clause for a discrete type, fixed point type, record type, or array type, specifying an Alignment value that is zero or a power of two, subject to the following:

Replace paragraph 31:

- An implementation need not support specified Alignments for combinations of Sizes and Alignments that cannot be easily loaded and stored by available machine instructions.

by:

- An implementation need not support an Alignment clause for a signed integer type specifying an Alignment greater than the largest Alignment value that is ever chosen by default by the implementation for any signed integer type. A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.

Replace paragraph 32:

- An implementation need not support specified Alignments that are greater than the maximum Alignment the implementation ever returns by default.

by:

- An implementation need not support a nonconfirming Alignment clause which could enable the creation of an object of an elementary type which cannot be easily loaded and stored by available machine instructions.
- An implementation need not support an Alignment specified for a derived tagged type which is not a multiple of the Alignment of the parent type. An implementation need not support a nonconfirming Alignment specified for a derived untagged by-reference type.

Delete paragraph 34:

- Same as above, for subtypes, but in addition:

Insert after paragraph 35:

- For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

the new paragraph:

- For other objects, an implementation should at least support the alignments supported for their subtype, subject to the following:

- An implementation need not support Alignments specified for objects of a by-reference type or for objects of types containing aliased subcomponents if the specified Alignment is not a multiple of the Alignment of the subtype of the object.

Delete paragraph 37:

4 The Alignment of a composite object is always equal to the least common multiple of the Alignments of its components, or a multiple thereof.

Replace paragraph 42:

The recommended level of support for the Size attribute of objects is:

by:

The size of an array object should not include its bounds.

The recommended level of support for the Size attribute of objects is the same as for subtypes (see below), except that only a confirming Size clause need be supported for an aliased elementary object.

Delete paragraph 43:

- A Size clause should be supported for an object if the specified Size is at least as large as its subtype's Size, and corresponds to a size in storage elements that is a multiple of the object's Alignment (if the Alignment is nonzero).

Replace paragraph 50:

If the Size of a subtype is specified, and allows for efficient independent addressability (see 9.10) on the target architecture, then the Size of the following objects of the subtype should equal the Size of the subtype:

by:

If the Size of a subtype allows for efficient independent addressability (see 9.10) on the target architecture, then the Size of the following objects of the subtype should equal the Size of the subtype:

Insert after paragraph 56:

- For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.

the new paragraph:

- An implementation should support a Size clause for a discrete type, fixed point type, record type, or array type, subject to the following:
  - An implementation need not support a Size clause for a signed integer type specifying a Size greater than that of the largest signed integer type supported by the implementation in the absence of a size clause (that is, when the size is chosen by default). A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.
  - A nonconfirming size clause for the first subtype of a derived untagged by-reference type need not be supported.

Replace paragraph 77:

8 The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Size, Component_Size, Alignment, External_Tag, Small, Bit_Order, Storage_Pool, Storage_Size, Write, Output, Read, Input, and Machine_Radix.

by:

8 The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Alignment, Bit_Order, Component_Size, External_Tag, Input, Machine_Radix, Output, Read, Size, Small, Storage_Pool, Storage_Size, Stream_Size, and Write.

Replace paragraph 85:

```
function My_Read(Stream : access Ada.Streams.Root_Stream_Type'Class)
  return T;
for T'Read use My_Read; -- see 13.13.2
```

by:

```
function My_Input(Stream : not null access Ada.Streams.Root_Stream_Type'Class)
  return T;
for T'Input use My_Input; -- see 13.13.2
```

13.4 Enumeration Representation Clauses

Replace paragraph 6:

The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type.

by:

Each component of the array_aggregate shall be given by an expression rather than a <>. The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type.

13.5.1 Record Representation Clauses

Replace paragraph 8:

The *first_subtype*_local_name of a record_representation_clause shall denote a specific nonlimited record or record extension subtype.

by:

The *first_subtype*_local_name of a record_representation_clause shall denote a specific record or record extension subtype.

Insert after paragraph 10:

The position, first_bit, and last_bit shall be static expressions. The value of position and first_bit shall be nonnegative. The value of last_bit shall be no less than first_bit - 1.

the new paragraphs:

If the nondefault bit ordering applies to the type, then either:

- the value of last_bit shall be less than the size of the largest machine scalar; or
- the value of first_bit shall be zero and the value of last_bit + 1 shall be a multiple of System.Storage_Unit.

Replace paragraph 13:

A record_representation_clause (without the mod_clause) specifies the layout. The storage place attributes (see 13.5.2) are taken from the values of the position, first_bit, and last_bit expressions after normalizing those values so that first_bit is less than Storage_Unit.

by:

A record_representation_clause (without the mod_clause) specifies the layout.

If the default bit ordering applies to the type, the position, first_bit, and last_bit of each component_clause directly specify the position and size of the corresponding component.

If the nondefault bit ordering applies to the type then the layout is determined as follows:

- the component_clauses for which the value of last_bit is greater than or equal to the size of the largest machine scalar directly specify the position and size of the corresponding component;

- for other component_clauses, all of the components having the same value of position are considered to be part of a single machine scalar, located at that position; this machine scalar has a size which is the smallest machine scalar size larger than the largest last_bit for all component_clauses at that position; the first_bit and last_bit of each component_clause are then interpreted as bit offsets in this machine scalar.

Insert after paragraph 17:

The recommended level of support for record_representation_clauses is:

the new paragraph:

- An implementation should support machine scalars that correspond to all of the integer, floating point, and address formats supported by the machine.

Replace paragraph 20:

- If the default bit ordering applies to the declaration of a given type, then for a component whose subtype's Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

by:

- For a component with a subtype whose Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

13.5.2 Storage Place Attributes

Replace paragraph 2:

R.C'Position
: Denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*.

by:

R.C'Position
: If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*.

Replace paragraph 3:

R.C'First_Bit
: Denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*.

by:

R.C'First_Bit
: If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*.

Replace paragraph 4:

R.C'Last_Bit
: Denotes the offset, from the start of the first of the storage elements 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*.

by:

R.C'Last_Bit
: If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the

start of the first of the storage elements 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*.

13.5.3 Bit Ordering

Replace paragraph 8:

- If Word_Size = Storage_Unit, then the implementation should support the nondefault bit ordering in addition to the default bit ordering.

by:

- The implementation should support the nondefault bit ordering in addition to the default bit ordering.

NOTES

13 Bit_Order clauses make it possible to write record_representation_clauses that can be ported between machines having different bit ordering. They do not guarantee transparent exchange of data between such machines.

13.7 The Package System

Replace paragraph 3:

```
package System is
   pragma Preelaborate(System);
```

by:

```
package System is
   pragma Pure(System);
```

In paragraph 15 replace:

```
   Default_Bit_Order : constant Bit_Order;
```

by:

```
   Default_Bit_Order : constant Bit_Order := implementation-defined;
```

Replace paragraph 34:

Address is of a definite, nonlimited type. Address represents machine addresses capable of addressing individual storage elements. Null_Address is an address that is distinct from the address of any object or program unit.

by:

Address is a definite, nonlimited type with preelaborable initialization (see 10.2.1). Address represents machine addresses capable of addressing individual storage elements. Null_Address is an address that is distinct from the address of any object or program unit.

Replace paragraph 35:

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order.

by:

Default_Bit_Order shall be a static constant. See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order.

Replace paragraph 36:

An implementation may add additional implementation-defined declarations to package System and its children. However, it is usually better for the implementation to provide additional functionality via implementation-defined children of System. Package System may be declared pure.

by:

An implementation may add additional implementation-defined declarations to package System and its children. However, it is usually better for the implementation to provide additional functionality via implementation-defined children of System.

13.7.1 The Package System.Storage_Elements

Replace paragraph 2:

```
package System.Storage_Elements is
   pragma Preelaborate(System.Storage_Elements);
```

by:

```
package System.Storage_Elements is
   pragma Pure(Storage_Elements);
```

Delete paragraph 15:

Package System.Storage_Elements may be declared pure.

13.7.2 The Package System.Address_To_Access_Conversions

Replace paragraph 5:

The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address. To_Pointer(X'Address) is equal to X'Unchecked_Access for any X that allows Unchecked_Access. To_Pointer(Null_Address) returns **null**. For other addresses, the behavior is unspecified. To_Address(**null**) returns Null_Address (for **null** of the appropriate type). To_Address(Y), where Y /= **null**, returns Y.**all**'Address.

by:

The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address. To_Pointer(X'Address) is equal to X'Unchecked_Access for any X that allows Unchecked_Access. To_Pointer(Null_Address) returns **null**. For other addresses, the behavior is unspecified. To_Address(**null**) returns Null_Address. To_Address(Y), where Y /= **null**, returns Y.**all**'Address.

13.8 Machine Code Insertions

Replace paragraph 10:

16 Machine code functions are exempt from the rule that a return_statement is required. In fact, return_statements are forbidden, since only code_statements are allowed.

by:

16 Machine code functions are exempt from the rule that a return statement is required. In fact, return statements are forbidden, since only code_statements are allowed.

13.9 Unchecked Type Conversions

Replace paragraph 11:

Otherwise, the effect is implementation defined; in particular, the result can be abnormal (see 13.9.1).

by:

Otherwise, if the result type is scalar, the result of the function is implementation defined, and can have an invalid representation (see 13.9.1). If the result type is nonscalar, the effect is implementation defined; in particular, the result can be abnormal (see 13.9.1).

Replace paragraph 14:

The Size of an array object should not include its bounds; hence, the bounds should not be part of the converted data.

by:

Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data.

13.9.1 Data Validity

Replace paragraph 6:

- The object is not scalar, and is passed to an **in out** or **out** parameter of an imported procedure or language-defined input procedure, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype.

by:

- The object is not scalar, and is passed to an **in out** or **out** parameter of an imported procedure, the Read procedure of an instance of Sequential_IO, Direct_IO, or Storage_IO, or the stream attribute T'Read, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype.
- The object is the return object of a function call of a nonscalar type, and the function is an imported function, an instance of Unchecked_Conversion, or the stream attribute T'Input, if after return from the function the representation of the return object does not represent a value of the function's subtype.

For an imported object, it is the programmer's responsibility to ensure that the object remains in a normal state.

Replace paragraph 12:

A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, and the result object has an invalid representation.

by:

A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, the result object has an invalid representation, and the result is used other than as the expression of an assignment_statement or an object_declaration, or as the prefix of a Valid attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a Valid attribute reference, is erroneous.

13.9.2 The Valid Attribute

Insert after paragraph 12:

20 X'Valid is not considered to be a read of X; hence, it is not an error to check the validity of invalid data.

the new paragraph:

22 The Valid attribute may be used to check the result of calling an instance of Unchecked_Conversion (or any other operation that can return invalid values). However, an exception handler should also be provided because implementations are permitted to raise Constraint_Error or Program_Error if they detect the use of an invalid representation (see 13.9.1).

13.11 Storage Management

Replace paragraph 2:

A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a *standard storage pool* for each access type. The user may define new pool types, and may override the choice of pool for an access type by specifying Storage_Pool for the type.

by:

A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a *standard storage pool* for each access-to-object type. The user may define new pool types, and may override the choice of pool for an access-to-object type by specifying Storage_Pool for the type.

Replace paragraph 6:

```
type Root_Storage_Pool is
    abstract new Ada.Finalization.Limited_Controlled with private;
```

by:

```
type Root_Storage_Pool is
    abstract new Ada.Finalization.Limited_Controlled with private;
pragma Preelaborable_Initialization(Root_Storage_Pool);
```

Replace paragraph 12:

For every access subtype S, the following representation attributes are defined:

by:

For every access-to-object subtype S, the following representation attributes are defined:

Replace paragraph 25:

A storage pool for an anonymous access type should be created at the point of an allocator for the type, and be reclaimed when the designated object becomes inaccessible.

by:

The storage pool used for an allocator of an anonymous access type should be determined as follows:

- If the allocator is defining a coextension (see 3.10.2) of an object being created by an outer allocator, then the storage pool used for the outer allocator should also be used for the coextension;
- For other access discriminants and access parameters, the storage pool should be created at the point of the allocator, and be reclaimed when the allocated object becomes inaccessible;
- Otherwise, a default storage pool should be created at the point where the anonymous access type is elaborated; such a storage pool need not support deallocation of individual objects.

13.11.1 The Max_Size_In_Storage_Elements Attribute

Replace paragraph 3:

Denotes the maximum value for Size_In_Storage_Elements that will be requested via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

by:

Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then this accounts for any calls on Allocate that could be performed to provide space for such coextensions. The value of this attribute is of type *universal_integer*.

13.11.2 Unchecked Storage Deallocation

Replace paragraph 9:

3. Free(X), when X is not equal to **null** first performs finalization, as described in 7.6. It then deallocates the storage occupied by the object designated by X. If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate, passing *access_to_variable_subtype_name*'Storage_Pool as the Pool parameter. Storage_Address is the value returned in the Storage_Address parameter of the corresponding Allocate call. Size_In_Storage_Elements and Alignment are the same values passed to the corresponding Allocate call. There is one exception: if the object being freed contains tasks, the object might not be deallocated.

by:

3. Free(X), when X is not equal to **null** first performs finalization of the object designated by X (and any coextensions of the object — see 3.10.2), as described in 7.6.1. It then deallocates the storage occupied by the object designated by X (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate, passing *access_to_variable_subtype_name*'Storage_Pool as the Pool parameter. Storage_Address is the value returned in the Storage_Address parameter of the corresponding

Allocate call. Size_In_Storage_Elements and Alignment are the same values passed to the corresponding Allocate call. There is one exception: if the object being freed contains tasks, the object might not be deallocated.

Replace paragraph 10:

After Free(X), the object designated by X, and any subcomponents thereof, no longer exist; their storage can be reused for other purposes.

by:

After Free(X), the object designated by X, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes.

13.12 Pragma Restrictions

Replace paragraph 4:

restriction ::= *restriction_*identifier
  | *restriction_parameter_*identifier => expression

by:

restriction ::= *restriction_*identifier
  | *restriction_parameter_*identifier => restriction_parameter_argument

restriction_parameter_argument ::= name | expression

Replace paragraph 7:

The set of restrictions is implementation defined.

by:

The set of restrictions is implementation defined.

Replace paragraph 10:

30 Restrictions intended to facilitate the construction of efficient tasking run-time systems are defined in D.7. Safety- and security-related restrictions are defined in H.4.

by:

30 Restrictions intended to facilitate the construction of efficient tasking run-time systems are defined in D.7. Restrictions intended for use when constructing high integrity systems are defined in H.4.

13.12.1 Language-Defined Restrictions

Insert new clause:

Static Semantics

The following *restriction_*identifiers are language-defined (additional restrictions are defined in the Specialized Needs Annexes):

No_Implementation_Attributes
: There are no implementation-defined attributes. This restriction applies only to the current compilation or environment, not the entire partition.

No_Implementation_Pragmas
: There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

No_Obsolescent_Features
: There is no use of language features defined in Annex J. It is implementation-defined if uses of the renamings of J.1 are detected by this restriction. This restriction applies only to the current compilation or environment, not the entire partition.

The following *restriction_parameter_*identifier is language defined:

No_Dependence
Specifies a library unit on which there are no semantic dependences.

Legality Rules

The restriction_parameter_argument of a No_Dependence restriction shall be a name; the name shall have the form of a full expanded name of a library unit, but need not denote a unit present in the environment.

Post-Compilation Rules

No compilation unit included in the partition shall depend semantically on the library unit identified by the name.

13.13.1 The Package Streams

Replace paragraph 3:

```
type Root_Stream_Type is abstract tagged limited private;
```

by:

```
type Root_Stream_Type is abstract tagged limited private;
pragma Preelaborable_Initialization(Root_Stream_Type);
```

Replace paragraph 8:

The Read operation transfers Item'Length stream elements from the specified stream to fill the array Item. The index of the last stream element transferred is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

by:

The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

Insert after paragraph 10:

See A.12.1, "The Package Streams.Stream_IO" for an example of extending type Root_Stream_Type.

the new paragraph:

If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error.

13.13.2 Stream-Oriented Attributes

Insert before paragraph 2:

For every subtype S of a specific type *T*, the following attributes are defined.

the new paragraphs:

For every subtype S of an elementary type *T*, the following representation attribute is defined:

S'Stream_Size
Denotes the number of bits occupied in a stream by items of subtype S. Hence, the number of stream elements required per item of elementary type *T* is:

```
T'Stream_Size / Ada.Streams.Stream_Element'Size
```

The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size.

Stream_Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, nonnegative, and a multiple of Stream_Element'Size.

Implementation Advice

If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

The recommended level of support for the Stream_Size attribute is:

- A Stream_Size clause should be supported for a discrete or fixed point type *T* if the specified Stream_Size is a multiple of Stream_Element'Size and is no less than the size of the first subtype of *T*, and no greater than the size of the largest type of the same elementary class (signed integer, modular integer, enumeration, ordinary fixed point, or decimal fixed point).

Replace paragraph 4:

```
procedure S'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class;
  Item : in T)
```

by:

```
procedure S'Write(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item : in T)
```

Replace paragraph 7:

```
procedure S'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class;
  Item : out T)
```

by:

```
procedure S'Read(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item : out T)
```

Replace paragraph 8.1:

For untagged derived types, the Write and Read attributes of the parent type are inherited as specified in 13.1; otherwise, the default implementations of these attributes are used. The default implementations of Write and Read attributes execute as follows:

by:

For an untagged derived type, the Write (resp. Read) attribute is inherited according to the rules given in 13.1 if the attribute is available for the parent type at the point where *T* is declared. For a tagged derived type, these attributes are not inherited, but rather the default implementations are used.

The default implementations of the Write and Read attributes, where available, execute as follows:

Replace paragraph 9:

For elementary types, the representation in terms of stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order, which is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of any ancestor type of *T* has been directly specified and the attribute of any ancestor type of the type of any of the extension components which are of a limited type has not been specified, the attribute of *T* shall be directly specified.

by:

For elementary types, Read reads (and Write writes) the number of stream elements implied by the Stream_Size for the type *T*; the representation of those stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order, which is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of the parent type or any progenitor type of *T* is available anywhere within the immediate scope of *T*, and the attribute of the parent type or the type

of any of the extension components is not available at the freezing point of *T*, then the attribute of *T* shall be directly specified.

Constraint_Error is raised by the predefined Write attribute if the value of the elementary item is outside the range of values representable using Stream_Size bits. For a signed integer type, an enumeration type, or a fixed point type, the range is unsigned only if the integer code for the lower bound of the first subtype is nonnegative, and a (symmetric) signed range that covers all values of the first subtype would require more than Stream_Size bits; otherwise the range is signed.

Replace paragraph 12:

```
procedure S'Class'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class;
  Item : in T'Class)
```

by:

```
procedure S'Class'Write(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item : in T'Class)
```

Replace paragraph 15:

```
procedure S'Class'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class;
  Item : out T'Class)
```

by:

```
procedure S'Class'Read(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item : out T'Class)
```

Delete paragraph 17:

If a stream element is the same size as a storage element, then the normal in-memory representation should be used by Read and Write for scalar objects. Otherwise, Read and Write should use the smallest number of stream elements needed to represent all values in the base range of the scalar type.

Replace paragraph 20:

```
procedure S'Output(
  Stream : access Ada.Streams.Root_Stream_Type'Class;
  Item : in T)
```

by:

```
procedure S'Output(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item : in T)
```

Replace paragraph 23:

```
function S'Input(
  Stream : access Ada.Streams.Root_Stream_Type'Class)
  return T
```

by:

```
function S'Input(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class)
  return T
```

Replace paragraph 25:

For untagged derived types, the Output and Input attributes of the parent type are inherited as specified in 13.1; otherwise, the default implementations of these attributes are used. The default implementations of Output and Input attributes execute as follows:

by:

For an untagged derived type, the Output (resp. Input) attribute is inherited according to the rules given in 13.1 if the attribute is available for the parent type at the point where *T* is declared. For a tagged derived type, these attributes are not inherited, but rather the default implementations are used.

The default implementations of the Output and Input attributes, where available, execute as follows:

Replace paragraph 27:

S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object (with the bounds or discriminants, if any, taken from the stream), initializes it with S'Read, and returns the value of the object.

by:

S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object (with the bounds or discriminants, if any, taken from the stream), passes it to S'Read, and returns the value of the object. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, and 7.6.1).

If *T* is an abstract type, then S'Input is an abstract function.

Replace paragraph 30:

```
procedure S'Class'Output(
  Stream : access Ada.Streams.Root_Stream_Type'Class;
  Item : in T'Class)
```

by:

```
procedure S'Class'Output(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item : in T'Class)
```

Replace paragraph 31:

First writes the external tag of *Item* to *Stream* (by calling String'Output(Tags.External_Tag(*Item*'Tag) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag.

by:

First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S.

Replace paragraph 33:

```
function S'Class'Input(
  Stream : access Ada.Streams.Root_Stream_Type'Class)
  return T'Class
```

by:

```
function S'Class'Input(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class)
  return T'Class
```

Replace paragraph 34:

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Internal_Tag(String'Input(*Stream*)) — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result.

by:

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which might raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that

result. If the specific type identified by the internal tag is not covered by *T*'Class or is abstract, Constraint_Error is raised.

Replace paragraph 35:

In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or whose component_declaration includes a default_expression, a check is made that the value returned by Read for the component belongs to its subtype. Constraint_Error is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by Read for the component is not a value of its subtype, Constraint_Error is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see 13.9.1).

by:

In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or whose component_declaration includes a default_expression, a check is made that the value returned by Read for the component belongs to its subtype. Constraint_Error is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by Read for the component is not a value of its subtype, Constraint_Error is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see 13.9.1). In the default implementation of Read for a composite type with defaulted discriminants, if the actual parameter of Read is constrained, a check is made that the discriminants read from the stream are equal to those of the actual parameter. Constraint_Error is raised if this check fails.

It is unspecified at which point and in which order these checks are performed. In particular, if Constraint_Error is raised due to the failure of one of these checks, it is unspecified how many stream elements have been read from the stream.

Replace paragraph 36:

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. All nonlimited types have default implementations for these operations. An attribute_reference for one of these attributes is illegal if the type is limited, unless the attribute has been specified by an attribute_definition_clause or (for a type extension) the attribute has been specified for an ancestor type. For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

by:

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. The subprogram name given in such a clause shall not denote an abstract subprogram. Furthermore, if a stream-oriented attribute is specified for an interface type by an attribute_definition_clause, the subprogram name given in the clause shall statically denote a null procedure.

A stream-oriented attribute for a subtype of a specific type *T* is *available* at places where one of the following conditions is true:

- *T* is nonlimited.
- The attribute_designator is Read (resp. Write) and *T* is a limited record extension, and the attribute Read (resp. Write) is available for the parent type of *T* and for the types of all of the extension components.
- *T* is a limited untagged derived type, and the attribute was inherited for the type.
- The attribute_designator is Input (resp. Output), and *T* is a limited type, and the attribute Read (resp. Write) is available for *T*.
- The attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible.

A stream-oriented attribute for a subtype of a class-wide type *T*'Class is available at places where one of the following conditions is true:

- *T* is nonlimited;

- the attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible; or
- the corresponding attribute of *T* is available, provided that if *T* has a partial view, the corresponding attribute is available at the end of the visible part where *T* is declared.

An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference. Furthermore, an attribute_reference for *T*'Input is illegal if *T* is an abstract type.

In the parameter_and_result_profiles for the stream-oriented attributes, the subtype of the Item parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. The same rule applies to the result of the Input attribute.

For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

A type is said to *support external streaming* if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation. A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of an access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an attribute_definition_clause, and that attribute_definition_clause is visible. An anonymous access type does not support external streaming. All other types support external streaming.

Erroneous Execution

If the internal tag returned by Descendant_Tag to T'Class'Input identifies a type that is not library-level and whose tag has not been created, or does not exist in the partition at the time of the call, execution is erroneous.

Insert after paragraph 36.1:

For every subtype *S* of a language-defined nonlimited specific type *T*, the output generated by S'Output or S'Write shall be readable by S'Input or S'Read, respectively. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex.

the new paragraphs:

If Constraint_Error is raised during a call to Read because of failure of one the above checks, the implementation must ensure that the discriminants of the actual parameter of Read are not modified.

Implementation Permissions

The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. However, all the calls on the Read and Write operations of the stream type needed to implement an explicit invocation of a stream-oriented attribute must take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see 12.3).

Replace paragraph 60:

```
procedure My_Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : My_Integer'Base);
```

by:

```
procedure My_Write(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item   : My_Integer'Base);
```

13.14 Freezing Rules

Insert after paragraph 7:

- The declaration of a record extension causes freezing of the parent subtype.

the new paragraph:

- The declaration of a record extension, interface type, task unit, or protected unit causes freezing of any progenitor types specified in the declaration.

Insert after paragraph 15:

- At the place where a subtype is frozen, its type is frozen. At the place where a type is frozen, any expressions or names within the full type definition cause freezing; the first subtype, and any component subtypes, index subtypes, and parent subtype of the type are frozen as well. For a specific tagged type, the corresponding class-wide type is frozen as well. For a class-wide type, the corresponding specific type is frozen as well.

the new paragraph:

- At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen.

Insert after paragraph 19:

An operational or representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.1).

the new paragraph:

Dynamic Semantics

The tag (see 3.9) of a tagged type T is created at the point where T is frozen.

Annex A: Predefined Language Environment

Replace paragraph 2:

Standard — A.1
  Ada — A.2
    Asynchronous_Task_Control — D.11
    Calendar — 9.6
    Characters — A.3.1
      Handling — A.3.2
      Latin_1 — A.3.3
    Command_Line — A.15
    Decimal — F.2
    Direct_IO — A.8.4
    Dynamic_Priorities — D.5
    Exceptions — 11.4.1
    Finalization — 7.6
    Float_Text_IO — A.10.9
    Float_Wide_Text_IO — A.11
    Integer_Text_IO — A.10.8
    Integer_Wide_Text_IO — A.11
    Interrupts — C.3.2
      Names — C.3.2
    IO_Exceptions — A.13
    Numerics — A.5
      Complex_Elementary_Functions — G.1.2
      Complex_Types — G.1.1
      Discrete_Random — A.5.2
      Elementary_Functions — A.5.1
      Float_Random — A.5.2
      Generic_Complex_Elementary_Functions — G.1.2
      Generic_Complex_Types — G.1.1
      Generic_Elementary_Functions — A.5.1
    Real_Time — D.8
    Sequential_IO — A.8.1
    Storage_IO — A.9
    Streams — 13.13.1
      Stream_IO — A.12.1

Standard (...continued)
  Ada (...continued)
    Strings — A.4.1
      Bounded — A.4.4
      Fixed — A.4.3
      Maps — A.4.2
        Constants — A.4.6
      Unbounded — A.4.5
      Wide_Bounded — A.4.7
      Wide_Fixed — A.4.7
      Wide_Maps — A.4.7
        Wide_Constants — A.4.7
      Wide_Unbounded — A.4.7
    Synchronous_Task_Control — D.10
    Tags — 3.9
    Task_Attributes — C.7.2
    Task_Identification — C.7.1
    Text_IO — A.10.1
      Complex_IO — G.1.3
      Editing — F.3.3
      Text_Streams — A.12.2
    Unchecked_Conversion — 13.9
    Unchecked_Deallocation — 13.11.2

Wide_Text_IO — A.11
Complex_IO — G.1.3
Editing — F.3.4
Text_Streams — A.12.3

Interfaces — B.2
C — B.3
Pointers — B.3.2
Strings — B.3.1
COBOL — B.4
Fortran — B.5

System — 13.7
Address_To_Access_Conversions — 13.7.2
Machine_Code — 13.8
RPC — E.5
Storage_Elements — 13.7.1
Storage_Pools — 13.11

by:

Standard — A.1
Ada — A.2
Assertions — 11.4.2
Asynchronous_Task_Control — D.11
Calendar — 9.6
Arithmetic — 9.6.1
Formatting — 9.6.1
Time_Zones — 9.6.1
Characters — A.3.1
Conversions — A.3.4
Handling — A.3.2
Latin_1 — A.3.3
Command_Line — A.15
Complex_Text_IO — G.1.3
Containers — A.18.1
Doubly_Linked_Lists — A.18.3
Generic_Array_Sort — A.18.16
Generic_Constrained_Array_Sort — A.18.16
Hashed_Maps — A.18.5
Hashed_Sets — A.18.8
Indefinite_Doubly_Linked_Lists — A.18.11
Indefinite_Hashed_Maps — A.18.12
Indefinite_Hashed_Sets — A.18.14
Indefinite_Ordered_Maps — A.18.13
Indefinite_Ordered_Sets — A.18.15
Indefinite_Vectors — A.18.10
Ordered_Maps — A.18.6
Ordered_Sets — A.18.9
Vectors — A.18.2
Decimal — F.2
Direct_IO — A.8.4
Directories — A.16
Information — A.16
Dispatching — D.2.1
EDF — D.2.6
Round_Robin — D.2.5
Dynamic_Priorities — D.5

Standard (...continued)
Ada (...continued)
Environment_Variables — A.17
Exceptions — 11.4.1
Execution_Time — D.14

Task_Attributes — C.7.2
Task_Identification — C.7.1
Task_Termination — C.7.3

Standard (...continued)
Ada (...continued)
Text_IO — A.10.1
Bounded_IO — A.10.11
Complex_IO — G.1.3
Editing — F.3.3
Text_Streams — A.12.2
Unbounded_IO — A.10.12
Unchecked_Conversion — 13.9
Unchecked_Deallocation — 13.11.2
Wide_Characters — A.3.1
Wide_Text_IO — A.11
Complex_IO — G.1.4
Editing — F.3.4
Text_Streams — A.12.3
Wide_Bounded_IO — A.11
Wide_Unbounded_IO — A.11
Wide_Wide_Characters — A.3.1
Wide_Wide_Text_IO — A.11
Complex_IO — G.1.5
Editing — F.3.5
Text_Streams — A.12.4
Wide_Wide_Bounded_IO — A.11
Wide_Wide_Unbounded_IO — A.11

Interfaces — B.2
C — B.3
Pointers — B.3.2
Strings — B.3.1
COBOL — B.4
Fortran — B.5

System — 13.7
Address_To_Access_Conversions — 13.7.2
Machine_Code — 13.8
RPC — E.5
Storage_Elements — 13.7.1
Storage_Pools — 13.11

Replace paragraph 3:

The implementation shall ensure that each language defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects.

by:

The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects.

A.1 The Package Standard

Replace paragraph 11:

- - *The integer type root_integer is predefined.*
- - *The corresponding universal type is universal_integer.*

by:

```
-- The integer type root_integer and the
-- corresponding universal type universal_integer are predefined.
```

Replace paragraph 20:

```
-- The floating point type root_real is predefined.
-- The corresponding universal type is universal_real.
```

by:

```
-- The floating point type root_real and the
-- corresponding universal type universal_real are predefined.
```

Insert after paragraph 34:

```
function "/" (Left : universal_fixed; Right : universal_fixed;)
  return universal_fixed;
```

the new paragraphs:

```
-- The type universal_access is predefined.
-- The following equality operators are predefined:

function "="  (Left, Right: universal_access) return Boolean;
function "/=" (Left, Right: universal_access) return Boolean;
```

In paragraph 35 replace:

```
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ',    --248 (16#F8#) .. 255 (16#FF#)
```

by:

```
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ');   --248 (16#F8#) .. 255 (16#FF#)
```

Replace paragraph 36:

```
-- The predefined operators for the type Character are the same as for
-- any enumeration type.

-- The declaration of type Wide_Character is based on the standard ISO 10646 BMP character set.
-- The first 256 positions have the same contents as type Character. See 3.5.2.

type Wide_Character is (nul, soh ... FFFE, FFFF);

package ASCII is ... end ASCII;  --Obsolescent; see J.5
```

by:

```
-- The predefined operators for the type Character are the same as for
-- any enumeration type.

-- The declaration of type Wide_Character is based on the standard ISO/IEC 10646:2003 BMP character
-- set. The first 256 positions have the same contents as type Character. See 3.5.2.

type Wide_Character is (nul, soh ... Hex_0000FFFE, Hex_0000FFFF);

-- The declaration of type Wide_Wide_Character is based on the full
-- ISO/IEC 10646:2003 character set. The first 65536 positions have the
-- same contents as type Wide_Character. See 3.5.2.

type Wide_Wide_Character is (nul, soh ... Hex_7FFFFFFE, Hex_7FFFFFFF);
for Wide_Wide_Character'Size use 32;

package ASCII is ... end ASCII;  --Obsolescent; see J.5
```

Replace paragraph 42:

```
-- The predefined operators for this type correspond to those for String
```

by:

```
-- The predefined operators for this type correspond to those for String.

type Wide_Wide_String is array (Positive range <>)
   of Wide_Wide_Character;
pragma Pack (Wide_Wide_String);

-- The predefined operators for this type correspond to those for String.
```

Replace paragraph 49:

In each of the types Character and Wide_Character, the character literals for the space character (position 32) and the non-breaking space character (position 160) correspond to different values. Unless indicated otherwise, each occurrence of the character literal ' ' in this International Standard refers to the space character. Similarly, the character literals for hyphen (position 45) and soft hyphen (position 173) correspond to different values. Unless indicated otherwise, each occurrence of the character literal '–' in this International Standard refers to the hyphen character.

by:

In each of the types Character, Wide_Character, and Wide_Wide_Character, the character literals for the space character (position 32) and the non-breaking space character (position 160) correspond to different values. Unless indicated otherwise, each occurrence of the character literal ' ' in this International Standard refers to the space character. Similarly, the character literals for hyphen (position 45) and soft hyphen (position 173) correspond to different values. Unless indicated otherwise, each occurrence of the character literal '–' in this International Standard refers to the hyphen character.

A.3 Character Handling

Replace paragraph 1:

This clause presents the packages related to character processing: an empty pure package Characters and child packages Characters.Handling and Characters.Latin_1. The package Characters.Handling provides classification and conversion functions for Character data, and some simple functions for dealing with Wide_Character data. The child package Characters.Latin_1 declares a set of constants initialized to values of type Character.

by:

This clause presents the packages related to character processing: an empty pure package Characters and child packages Characters.Handling and Characters.Latin_1. The package Characters.Handling provides classification and conversion functions for Character data, and some simple functions for dealing with Wide_Character and Wide_Wide_Character data. The child package Characters.Latin_1 declares a set of constants initialized to values of type Character.

A.3.1 Packages Characters, Wide_Characters, and Wide_Wide_Characters

Replace the title:

The Package Characters

by:

The Packages Characters, Wide_Characters, and Wide_Wide_Characters

Insert after paragraph 2:

```
package Ada.Characters is
   pragma Pure(Characters);
end Ada.Characters;
```

the new paragraphs:

The library package Wide_Characters has the following declaration:

```
package Ada.Wide_Characters is
   pragma Pure(Wide_Characters);
end Ada.Wide_Characters;
```

The library package Wide_Wide_Characters has the following declaration:

```
package Ada.Wide_Wide_Characters is
   pragma Pure(Wide_Wide_Characters);
end Ada.Wide_Wide_Characters;
```

Implementation Advice

If an implementation chooses to provide implementation-defined operations on Wide_Character or Wide_String (such as case mapping, classification, collating and sorting, etc.) it should do so by providing child units of Wide_Characters. Similarly if it chooses to provide implementation-defined operations on Wide_Wide_Character or Wide_Wide_String it should do so by providing child units of Wide_Wide_Characters.

A.3.2 The Package Characters.Handling

Replace paragraph 2:

```
package Ada.Characters.Handling is
   pragma Preelaborate(Handling);
```

by:

```
with Ada.Characters.Conversions;
package Ada.Characters.Handling is
   pragma Pure(Handling);
```

Replace paragraph 13:

```
-- Classifications of and conversions between Wide_Character and Character.
```

by:

```
-- The functions Is_Character, Is_String, To_Character, To_String, To_Wide_Character,
-- and To_Wide_String are obsolescent; see J.14.
```

Delete paragraph 14:

```
function Is_Character (Item : in Wide_Character) return Boolean;
function Is_String    (Item : in Wide_String)    return Boolean;
```

Delete paragraph 15:

```
function To_Character (Item       : in Wide_Character;
                       Substitute : in Character := ' ')
   return Character;
```

Delete paragraph 16:

```
function To_String    (Item       : in Wide_String;
                       Substitute : in Character := ' ')
   return String;
```

Delete paragraph 17:

```
function To_Wide_Character (Item : in Character) return Wide_Character;
```

Delete paragraph 18:

```
function To_Wide_String    (Item : in String)    return Wide_String;
```

Delete paragraph 42:

The following set of functions test Wide_Character values for membership in Character, or convert between corresponding characters of Wide_Character and Character.

Delete paragraph 43:

Is_Character
: Returns True if Wide_Character'Pos(Item) <= Character'Pos(Character'Last).

Delete paragraph 44:

Is_String
: Returns True if Is_Character(Item(I)) is True for each I in Item'Range.

Delete paragraph 45:

To_Character
: Returns the Character corresponding to Item if Is_Character(Item), and returns the Substitute Character otherwise.

Delete paragraph 46:

To_String
: Returns the String whose range is 1..Item'Length and each of whose elements is given by To_Character of the corresponding element in Item.

Delete paragraph 47:

To_Wide_Character
: Returns the Wide_Character X such that Character'Pos(Item) = Wide_Character'Pos(X).

Delete paragraph 48:

To_Wide_String
: Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item.

Delete paragraph 49:

If an implementation provides a localized definition of Character or Wide_Character, then the effects of the subprograms in Characters.Handling should reflect the localizations. See also 3.5.2.

A.3.4 The Package Characters.Conversions

Insert new clause:

The library package Characters.Conversions has the following declaration:

```
package Ada.Characters.Conversions is
   pragma Pure(Conversions);

   function Is_Character (Item : in Wide_Character)      return Boolean;
   function Is_String    (Item : in Wide_String)         return Boolean;
   function Is_Character (Item : in Wide_Wide_Character) return Boolean;
   function Is_String    (Item : in Wide_Wide_String)    return Boolean;
   function Is_Wide_Character (Item : in Wide_Wide_Character)
      return Boolean;
   function Is_Wide_String    (Item : in Wide_Wide_String)
      return Boolean;

   function To_Wide_Character (Item : in Character) return Wide_Character;
   function To_Wide_String    (Item : in String)    return Wide_String;
   function To_Wide_Wide_Character (Item : in Character)
      return Wide_Wide_Character;
   function To_Wide_Wide_String    (Item : in String)
      return Wide_Wide_String;
   function To_Wide_Wide_Character (Item : in Wide_Character)
      return Wide_Wide_Character;
   function To_Wide_Wide_String    (Item : in Wide_String)
      return Wide_Wide_String;

   function To_Character (Item       : in Wide_Character;
                          Substitute : in Character := ' ')
      return Character;
```

```
   function To_String    (Item       : in Wide_String;
                          Substitute : in Character := ' ')
      return String;
   function To_Character (Item :       in Wide_Wide_Character;
                          Substitute : in Character := ' ')
      return Character;
   function To_String    (Item :       in Wide_Wide_String;
                          Substitute : in Character := ' ')
      return String;
   function To_Wide_Character (Item :       in Wide_Wide_Character;
                               Substitute : in Wide_Character := ' ')
      return Wide_Character;
   function To_Wide_String    (Item :       in Wide_Wide_String;
                               Substitute : in Wide_Character := ' ')
      return Wide_String;

end Ada.Characters.Conversions;
```

The functions in package Characters.Conversions test Wide_Wide_Character or Wide_Character values for membership in Wide_Character or Character, or convert between corresponding characters of Wide_Wide_Character, Wide_Character, and Character.

```
function Is_Character (Item : in Wide_Character) return Boolean;
```

Returns True if Wide_Character'Pos(Item) <= Character'Pos(Character'Last).

```
function Is_Character (Item : in Wide_Wide_Character) return Boolean;
```

Returns True if Wide_Wide_Character'Pos(Item) <= Character'Pos(Character'Last).

```
function Is_Wide_Character (Item : in Wide_Wide_Character) return Boolean;
```

Returns True if Wide_Wide_Character'Pos(Item) <= Wide_Character'Pos(Wide_Character'Last).

```
function Is_String (Item : in Wide_String)      return Boolean;
function Is_String (Item : in Wide_Wide_String) return Boolean;
```

Returns True if Is_Character(Item(I)) is True for each I in Item'Range.

```
function Is_Wide_String (Item : in Wide_Wide_String) return Boolean;
```

Returns True if Is_Wide_Character(Item(I)) is True for each I in Item'Range.

```
function To_Character (Item :       in Wide_Character;
                       Substitute : in Character := ' ') return Character;
function To_Character (Item :       in Wide_Wide_Character;
                       Substitute : in Character := ' ') return Character;
```

Returns the Character corresponding to Item if Is_Character(Item), and returns the Substitute Character otherwise.

```
function To_Wide_Character (Item : in Character) return Wide_Character;
```

Returns the Wide_Character X such that Character'Pos(Item) = Wide_Character'Pos (X).

```
function To_Wide_Character (Item :       in Wide_Wide_Character;
                            Substitute : in Wide_Character := ' ')
   return Wide_Character;
```

Returns the Wide_Character corresponding to Item if Is_Wide_Character(Item), and returns the Substitute Wide_Character otherwise.

```
function To_Wide_Wide_Character (Item : in Character)
   return Wide_Wide_Character;
```

Returns the Wide_Wide_Character X such that Character'Pos(Item) = Wide_Wide_Character'Pos (X).

```
function To_Wide_Wide_Character (Item : in Wide_Character)
   return Wide_Wide_Character;
```

Returns the Wide_Wide_Character X such that Wide_Character'Pos(Item) = Wide_Wide_Character'Pos (X).

```
function To_String (Item :       in Wide_String;
                    Substitute : in Character := ' ') return String;
function To_String (Item :       in Wide_Wide_String;
                    Substitute : in Character := ' ') return String;
```

Returns the String whose range is 1..Item'Length and each of whose elements is given by To_Character of the corresponding element in Item.

```
function To_Wide_String (Item : in String) return Wide_String;
```

Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item.

```
function To_Wide_String (Item :       in Wide_Wide_String;
                         Substitute : in Wide_Character := ' ')
   return Wide_String;
```

Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item with the given Substitute Wide_Character.

```
function To_Wide_Wide_String (Item : in String) return Wide_Wide_String;
function To_Wide_Wide_String (Item : in Wide_String)
   return Wide_Wide_String;
```

Returns the Wide_Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Wide_Character of the corresponding element in Item.

A.4 String Handling

Replace paragraph 1:

This clause presents the specifications of the package Strings and several child packages, which provide facilities for dealing with string data. Fixed-length, bounded-length, and unbounded-length strings are supported, for both String and Wide_String. The string-handling subprograms include searches for pattern strings and for characters in program-specified sets, translation (via a character-to-character mapping), and transformation (replacing, inserting, overwriting, and deleting of substrings).

by:

This clause presents the specifications of the package Strings and several child packages, which provide facilities for dealing with string data. Fixed-length, bounded-length, and unbounded-length strings are supported, for String, Wide_String, and Wide_Wide_String. The string-handling subprograms include searches for pattern strings and for characters in program-specified sets, translation (via a character-to-character mapping), and transformation (replacing, inserting, overwriting, and deleting of substrings).

A.4.1 The Package Strings

Replace paragraph 4:

```
Space      : constant Character      := ' ';
Wide_Space : constant Wide_Character := ' ';
```

by:

```
Space      : constant Character      := ' ';
Wide_Space : constant Wide_Character := ' ';
Wide_Wide_Space : constant Wide_Wide_Character := ' ';
```

A.4.2 The Package Strings.Maps

Replace paragraph 3:

```
package Ada.Strings.Maps is
   pragma Preelaborate(Maps);
```

by:

```
package Ada.Strings.Maps is
   pragma Pure(Maps);
```

Replace paragraph 4:

```
   -- Representation for a set of character values:
   type Character_Set is private;
```

by:

```
   -- Representation for a set of character values:
   type Character_Set is private;
   pragma Preelaborable_Initialization(Character_Set);
```

Replace paragraph 20:

```
   -- Representation for a character to character mapping:
   type Character_Mapping is private;
```

by:

```
   -- Representation for a character to character mapping:
   type Character_Mapping is private;
   pragma Preelaborable_Initialization(Character_Mapping);
```

A.4.3 Fixed-Length String Handling

Insert after paragraph 8:

```
-- Search subprograms
```

the new paragraphs:

```
   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;

   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```

Insert after paragraph 10:

```
   function Index (Source  : in String;
                   Pattern : in String;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```

the new paragraph:

```
   function Index (Source  : in String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going   : in Direction := Forward)
      return Natural;
```

Insert after paragraph 11:

```
   function Index (Source  : in String;
                   Set     : in Maps.Character_Set;
```

```
                     Test    : in Membership := Inside;
                     Going   : in Direction := Forward)
         return Natural;
```

the new paragraph:

```
      function Index_Non_Blank (Source : in String;
                                From   : in Positive;
                                Going  : in Direction := Forward)
         return Natural;
```

Insert after paragraph 56:

- Otherwise, Length_Error is propagated.

the new paragraphs:

```
   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;

   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```

Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated.

Replace paragraph 58:

Each Index function searches for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If Going = Forward, then Index returns the smallest index I such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern. If there is no such slice, then 0 is returned. If Pattern is the null string then Pattern_Error is propagated.

by:

If Going = Forward, returns

```
Index (Source, Pattern, Source'First, Forward, Mapping);
```

otherwise returns

```
Index (Source, Pattern, Source'Last, Backward, Mapping);
```

```
   function Index (Source  : in String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going   : in Direction := Forward)
      return Natural;
```

Index searches for the first or last occurrence of any of a set of characters (when Test=Inside), or any of the complement of a set of characters (when Test=Outside). If From is not in Source'Range, then Index_Error is propagated. Otherwise, it returns the smallest index I >= From (if Going=Forward) or the largest index I <=

From (if Going=Backward) such that Source(I) satisfies the Test condition with respect to Set; it returns 0 if there is no such Character in Source.

Replace paragraph 60:

Index searches for the first or last occurrence of any of a set of characters (when Test=Inside), or any of the complement of a set of characters (when Test=Outside). It returns the smallest index I (if Going=Forward) or the largest index I (if Going=Backward) such that Source(I) satisfies the Test condition with respect to Set; it returns 0 if there is no such Character in Source.

by:

If Going = Forward, returns

```
Index (Source, Set, Source'First, Test, Forward);
```

otherwise returns

```
Index (Source, Set, Source'Last, Test, Backward);
```

```
function Index_Non_Blank (Source : in String;
                          From   : in Positive;
                          Going  : in Direction := Forward)
   return Natural;
```

Returns Index (Source, Maps.To_Set(Space), From, Outside, Going);

A.4.4 Bounded-Length String Handling

Insert after paragraph 12:

```
function To_String (Source : in Bounded_String) return String;
```

the new paragraphs:

```
procedure Set_Bounded_String
   (Target :    out Bounded_String;
    Source : in     String;
    Drop   : in     Truncation := Error);
```

Insert after paragraph 28:

```
function Slice (Source : in Bounded_String;
                Low    : in Positive;
                High   : in Natural)
   return String;
```

the new paragraphs:

```
function Bounded_Slice
   (Source : in Bounded_String;
    Low    : in Positive;
    High   : in Natural)
       return Bounded_String;

procedure Bounded_Slice
   (Source : in     Bounded_String;
    Target :    out Bounded_String;
    Low    : in     Positive;
    High   : in     Natural);
```

Replace paragraph 43:

-- *Search functions*

by:

-- *Search subprograms*

```
function Index (Source  : in Bounded_String;
```

```
                        Pattern : in String;
                        From    : in Positive;
                        Going   : in Direction := Forward;
                        Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;

   function Index (Source  : in Bounded_String;
                        Pattern : in String;
                        From    : in Positive;
                        Going   : in Direction := Forward;
                        Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```

Insert after paragraph 45:

```
   function Index (Source  : in Bounded_String;
                        Pattern : in String;
                        Going   : in Direction := Forward;
                        Mapping : in Maps.Character_Mapping_Function)
      return Natural;
```

the new paragraph:

```
   function Index (Source  : in Bounded_String;
                        Set     : in Maps.Character_Set;
                        From    : in Positive;
                        Test    : in Membership := Inside;
                        Going   : in Direction := Forward)
      return Natural;
```

Insert after paragraph 46:

```
   function Index (Source  : in Bounded_String;
                        Set     : in Maps.Character_Set;
                        Test    : in Membership := Inside;
                        Going   : in Direction := Forward)
      return Natural;
```

the new paragraph:

```
   function Index_Non_Blank (Source : in Bounded_String;
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
```

Insert after paragraph 92:

To_String returns the String value with lower bound 1 represented by Source. If B is a Bounded_String, then B = To_Bounded_String(To_String(B)).

the new paragraphs:

```
procedure Set_Bounded_String
   (Target :    out Bounded_String;
    Source : in     String;
    Drop   : in     Truncation := Error);
```

Equivalent to Target := To_Bounded_String (Source, Drop);

Replace paragraph 101:

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source).

by:

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). The bounds of the returned string are Low and High.

```
function Bounded_Slice
```

```
(Source : in Bounded_String;
 Low    : in Positive;
 High   : in Natural)
    return Bounded_String;
```

Returns the slice at positions Low through High in the string represented by Source as a bounded string; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source).

```
procedure Bounded_Slice
   (Source : in     Bounded_String;
    Target :    out Bounded_String;
    Low    : in     Positive;
    High   : in     Natural);
```

Equivalent to Target := Bounded_Slice (Source, Low, High);

A.4.5 Unbounded-Length String Handling

Replace paragraph 4:

```
type Unbounded_String is private;
```

by:

```
type Unbounded_String is private;
pragma Preelaborable_Initialization(Unbounded_String);
```

Insert after paragraph 11:

```
function To_String (Source : in Unbounded_String) return String;
```

the new paragraphs:

```
procedure Set_Unbounded_String
   (Target :    out Unbounded_String;
    Source : in     String);
```

Insert after paragraph 22:

```
function Slice (Source : in Unbounded_String;
                Low    : in Positive;
                High   : in Natural)
   return String;
```

the new paragraphs:

```
function Unbounded_Slice
    (Source : in Unbounded_String;
     Low    : in Positive;
     High   : in Natural)
        return Unbounded_String;

 procedure Unbounded_Slice
    (Source : in     Unbounded_String;
     Target :    out Unbounded_String;
     Low    : in     Positive;
     High   : in     Natural);
```

Insert after paragraph 38:

```
-- Search subprograms
```

the new paragraphs:

```
function Index (Source  : in Unbounded_String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;
```

```
function Index (Source  : in Unbounded_String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping_Function)
   return Natural;
```

Insert after paragraph 40:

```
function Index (Source  : in Unbounded_String;
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping_Function)
   return Natural;
```

the new paragraph:

```
function Index (Source  : in Unbounded_String;
                Set     : in Maps.Character_Set;
                From    : in Positive;
                Test    : in Membership := Inside;
                Going    : in Direction := Forward)
   return Natural;
```

Insert after paragraph 41:

```
function Index (Source  : in Unbounded_String;
                Set     : in Maps.Character_Set;
                Test    : in Membership := Inside;
                Going    : in Direction := Forward)
   return Natural;
```

the new paragraph:

```
function Index_Non_Blank (Source : in Unbounded_String;
                          From   : in Positive;
                          Going  : in Direction := Forward)
   return Natural;
```

Insert after paragraph 72:

```
private
   ... -- not specified by the language
end Ada.Strings.Unbounded;
```

the new paragraph:

The type Unbounded_String needs finalization (see 7.6).

Insert after paragraph 79:

- If U is an Unbounded_String, then To_Unbounded_String(To_String(U)) = U.

the new paragraph:

The procedure Set_Unbounded_String sets Target to an Unbounded_String that represents Source.

Insert after paragraph 82:

The Element, Replace_Element, and Slice subprograms have the same effect as the corresponding bounded-length string subprograms.

the new paragraph:

The function Unbounded_Slice returns the slice at positions Low through High in the string represented by Source as an Unbounded_String. The procedure Unbounded_Slice sets Target to the Unbounded_String representing the slice at positions Low through High in the string represented by Source. Both routines propagate Index_Error if Low > Length(Source)+1 or High > Length(Source).

A.4.6 String-Handling Sets and Mappings

Replace paragraph 3:

```
package Ada.Strings.Maps.Constants is
   pragma Preelaborate(Constants);
```

by:

```
package Ada.Strings.Maps.Constants is
   pragma Pure(Constants);
```

A.4.7 Wide_String Handling

Replace paragraph 1:

Facilities for handling strings of Wide_Character elements are found in the packages Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_Unbounded, and Strings.Wide_Maps.Wide_Constants. They provide the same string-handling operations as the corresponding packages for strings of Character elements.

by:

Facilities for handling strings of Wide_Character elements are found in the packages Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_Unbounded, and Strings.Wide_Maps.Wide_Constants, and in the functions Strings.Wide_Hash, Strings.Wide_Fixed.Wide_Hash, Strings.Wide_Bounded.Wide_Hash, and Strings.Wide_Unbounded.Wide_Hash. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Replace paragraph 4:

```
-- Representation for a set of Wide_Character values:
type Wide_Character_Set is private;
```

by:

```
-- Representation for a set of Wide_Character values:
type Wide_Character_Set is private;
pragma Preelaborable_Initialization(Wide_Character_Set);
```

Replace paragraph 20:

```
-- Representation for a Wide_Character to Wide_Character mapping:
type Wide_Character_Mapping is private;
```

by:

```
-- Representation for a Wide_Character to Wide_Character mapping:
type Wide_Character_Mapping is private;
pragma Preelaborable_Initialization(Wide_Character_Mapping);
```

Replace paragraph 29:

For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants the corresponding wide string package has the same contents except that

by:

For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants, and for functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, and Strings.Unbounded.Hash, the corresponding wide string package has the same contents except that

Insert after paragraph 40:

- To_Wide_String replaces To_String

the new paragraphs:

- Set_Bounded_Wide_String replaces Set_Bounded_String

Insert after paragraph 44:

- To_Unbounded_Wide_String replaces To_Unbounded_String

the new paragraphs:

- Set_Unbounded_Wide_String replaces Set_Unbounded_String

Replace paragraph 46:

```
Character_Set : constant Wide_Maps.Wide_Character_Set;
--Contains each Wide_Character value WC such that Characters.Is_Character(WC) is True
```

by:

```
Character_Set : constant Wide_Maps.Wide_Character_Set;
--Contains each Wide_Character value WC such that
--Characters.Conversions.Is_Character(WC) is True
```

Each Wide_Character_Set constant in the package Strings.Wide_Maps.Wide_Constants contains no values outside the Character portion of Wide_Character. Similarly, each Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Character.

Pragma Pure is replaced by pragma Preelaborate in Strings.Wide_Maps.Wide_Constants.

Delete paragraph 48:

13 Each Wide_Character_Set constant in the package Strings.Wide_Maps.Wide_Constants contains no values outside the Character portion of Wide_Character. Similarly, each Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Character.

A.4.8 Wide_Wide_String Handling

Insert new clause:

Facilities for handling strings of Wide_Wide_Character elements are found in the packages Strings.Wide_Wide_Maps, Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, Strings.Wide_Wide_Unbounded, and Strings.Wide_Wide_Maps.Wide_Wide_Constants, and in the functions Strings.Wide_Wide_Hash, Strings.Wide_Wide_Fixed.Wide_Wide_Hash, Strings.Wide_Wide_Bounded.Wide_Wide_Hash, and Strings.Wide_Wide_Unbounded.Wide_Wide_Hash. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Static Semantics

The library package Strings.Wide_Wide_Maps has the following declaration.

```
package Ada.Strings.Wide_Wide_Maps is
   pragma Preelaborate(Wide_Wide_Maps);

   -- Representation for a set of Wide_Wide_Character values:
   type Wide_Wide_Character_Set is private;
   pragma Preelaborable_Initialization(Wide_Wide_Character_Set);

   Null_Set : constant Wide_Wide_Character_Set;

   type Wide_Wide_Character_Range is
      record
         Low  : Wide_Wide_Character;
         High : Wide_Wide_Character;
      end record;
   -- Represents Wide_Wide_Character range Low..High

   type Wide_Wide_Character_Ranges is array (Positive range <>)
         of Wide_Wide_Character_Range;

   function To_Set (Ranges : in Wide_Wide_Character_Ranges)
         return Wide_Wide_Character_Set;
```

```
function To_Set (Span : in Wide_Wide_Character_Range)
      return Wide_Wide_Character_Set;

function To_Ranges (Set : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Ranges;

function "=" (Left, Right : in Wide_Wide_Character_Set) return Boolean;

function "not" (Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "and" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "or" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "xor" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "-" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;

function Is_In (Element : in Wide_Wide_Character;
                Set     : in Wide_Wide_Character_Set)
      return Boolean;

function Is_Subset (Elements : in Wide_Wide_Character_Set;
                    Set      : in Wide_Wide_Character_Set)
      return Boolean;

function "<=" (Left  : in Wide_Wide_Character_Set;
               Right : in Wide_Wide_Character_Set)
      return Boolean renames Is_Subset;

-- Alternative representation for a set of Wide_Wide_Character values:
subtype Wide_Wide_Character_Sequence is Wide_Wide_String;

function To_Set (Sequence : in Wide_Wide_Character_Sequence)
      return Wide_Wide_Character_Set;

function To_Set (Singleton : in Wide_Wide_Character)
      return Wide_Wide_Character_Set;

function To_Sequence (Set : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Sequence;

-- Representation for a Wide_Wide_Character to Wide_Wide_Character
-- mapping:
type Wide_Wide_Character_Mapping is private;
pragma Preelaborable_Initialization(Wide_Wide_Character_Mapping);

function Value (Map     : in Wide_Wide_Character_Mapping;
                Element : in Wide_Wide_Character)
      return Wide_Wide_Character;

Identity : constant Wide_Wide_Character_Mapping;

function To_Mapping (From, To : in Wide_Wide_Character_Sequence)
      return Wide_Wide_Character_Mapping;

function To_Domain (Map : in Wide_Wide_Character_Mapping)
      return Wide_Wide_Character_Sequence;

function To_Range (Map : in Wide_Wide_Character_Mapping)
      return Wide_Wide_Character_Sequence;

type Wide_Wide_Character_Mapping_Function is
      access function (From : in Wide_Wide_Character)
```

```
        return Wide_Wide_Character;

private
   ... -- not specified by the language
end Ada.Strings.Wide_Wide_Maps;
```

The context clause for each of the packages Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, and Strings.Wide_Wide_Unbounded identifies Strings.Wide_Wide_Maps instead of Strings.Maps.

For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants, and for functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, and Strings.Unbounded.Hash, the corresponding wide wide string package or function has the same contents except that

- Wide_Wide_Space replaces Space
- Wide_Wide_Character replaces Character
- Wide_Wide_String replaces String
- Wide_Wide_Character_Set replaces Character_Set
- Wide_Wide_Character_Mapping replaces Character_Mapping
- Wide_Wide_Character_Mapping_Function replaces Character_Mapping_Function
- Wide_Wide_Maps replaces Maps
- Bounded_Wide_Wide_String replaces Bounded_String
- Null_Bounded_Wide_Wide_String replaces Null_Bounded_String
- To_Bounded_Wide_Wide_String replaces To_Bounded_String
- To_Wide_Wide_String replaces To_String
- Set_Bounded_Wide_Wide_String replaces Set_Bounded_String
- Unbounded_Wide_Wide_String replaces Unbounded_String
- Null_Unbounded_Wide_Wide_String replaces Null_Unbounded_String
- Wide_Wide_String_Access replaces String_Access
- To_Unbounded_Wide_Wide_String replaces To_Unbounded_String
- Set_Unbounded_Wide_Wide_String replaces Set_Unbounded_String

The following additional declarations are present in Strings.Wide_Wide_Maps.Wide_Wide_Constants:

```
Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
-- Contains each Wide_Wide_Character value WWC such that
-- Characters.Conversions.Is_Character(WWC) is True
Wide_Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
-- Contains each Wide_Wide_Character value WWC such that
-- Characters.Conversions.Is_Wide_Character(WWC) is True
```

Each Wide_Wide_Character_Set constant in the package Strings.Wide_Wide_Maps.Wide_Wide_Constants contains no values outside the Character portion of Wide_Wide_Character. Similarly, each Wide_Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Wide_Character.

Pragma Pure is replaced by pragma Preelaborate in Strings.Wide_Wide_Maps.Wide_Wide_Constants.

NOTES

14 If a null Wide_Wide_Character_Mapping_Function is passed to any of the Wide_Wide_String handling subprograms, Constraint_Error is propagated.

A.4.9 String Hashing

Insert new clause:

Static Semantics

The library function Strings.Hash has the following declaration:

```
with Ada.Containers;
function Ada.Strings.Hash (Key : String) return Containers.Hash_Type;
pragma Pure(Hash);
```

Returns an implementation-defined value which is a function of the value of Key. If *A* and *B* are strings such that *A* equals *B*, Hash(*A*) equals Hash(*B*).

The library function Strings.Fixed.Hash has the following declaration:

```
with Ada.Containers, Ada.Strings.Hash;
function Ada.Strings.Fixed.Hash (Key : String) return Containers.Hash_Type
   renames Ada.Strings.Hash;
pragma Pure(Hash);
```

The generic library function Strings.Bounded.Hash has the following declaration:

```
with Ada.Containers;
generic
   with package Bounded is
                     new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash (Key : Bounded.Bounded_String)
   return Containers.Hash_Type;
pragma Preelaborate(Hash);
```

Strings.Bounded.Hash is equivalent to the function call Strings.Hash (Bounded.To_String (Key));

The library function Strings.Unbounded.Hash has the following declaration:

```
with Ada.Containers;
function Ada.Strings.Unbounded.Hash (Key : Unbounded_String)
   return Containers.Hash_Type;
pragma Preelaborate(Hash);
```

Strings.Unbounded.Hash is equivalent to the function call Strings.Hash (To_String (Key));

Implementation Advice

The Hash functions should be good hash functions, returning a wide spread of values for different string values. It should be unlikely for similar strings to return the same value.

A.5 The Numerics Packages

Replace paragraph 3:

```
package Ada.Numerics is
   pragma Pure(Numerics);
   Argument_Error : exception;
   Pi : constant :=
          3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
   e  : constant :=
          2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

by:

```
package Ada.Numerics is
   pragma Pure(Numerics);
   Argument_Error : exception;
   Pi : constant :=
          3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
   π : constant := Pi;
   e  : constant :=
```

```
            2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

A.5.2 Random Number Generation

Insert after paragraph 15:

```
private
   ... -- not specified by the language
end Ada.Numerics.Float_Random;
```

the new paragraph:

The type Generator needs finalization (see 7.6).

Insert after paragraph 27:

```
private
   ... -- not specified by the language
end Ada.Numerics.Discrete_Random;
```

the new paragraph:

The type Generator needs finalization (see 7.6) in every instantiation of Numerics.Discrete_Random.

Replace paragraph 53:

```
-Log(Random(G) + Float'Model_Small))
```

by:

```
-Log(Random(G) + Float'Model_Small)
```

A.5.3 Attributes of Floating Point Types

Insert after paragraph 41:

The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

the new paragraphs:

S'Machine_Rounding

S'Machine_Rounding denotes a function with the following specification:

```
function S'Machine_Rounding (X : T)
     return T
```

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.

A.6 Input-Output

Replace paragraph 1:

Input-output is provided through language-defined packages, each of which is a child of the root package Ada. The generic packages Sequential_IO and Direct_IO define input-output operations applicable to files containing elements of a given type. The generic package Storage_IO supports reading from and writing to an in-memory buffer. Additional operations for text input-output are supplied in the packages Text_IO and Wide_Text_IO. Heterogeneous input-output is provided through the child packages Streams.Stream_IO and Text_IO.Text_Streams (see also 13.13). The package IO_Exceptions defines the exceptions needed by the predefined input-output packages.

by:

Input-output is provided through language-defined packages, each of which is a child of the root package Ada. The generic packages Sequential_IO and Direct_IO define input-output operations applicable to files containing elements of a given type. The generic package Storage_IO supports reading from and writing to an in-memory buffer. Additional operations for text input-output are supplied in the packages Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. Heterogeneous input-output is provided through the child packages Streams.Stream_IO and Text_IO.Text_Streams (see also 13.13). The package IO_Exceptions defines the exceptions needed by the predefined input-output packages.

A.7 External Files and File Objects

Replace paragraph 4:

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) packages Text_IO for Character and String data, and Wide_Text_IO for Wide_Character and Wide_String data. Input-output for files containing streams of elements representing values of possibly different types is defined by means of the (nongeneric) package Streams.Stream_IO.

by:

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) packages Text_IO for Character and String data, Wide_Text_IO for Wide_Character and Wide_String data, and Wide_Wide_Text_IO for Wide_Wide_Character and Wide_Wide_String data. Input-output for files containing streams of elements representing values of possibly different types is defined by means of the (nongeneric) package Streams.Stream_IO.

Replace paragraph 10:

```
type File_Mode is (In_File, Out_File, Append_File);
--  for Sequential_IO, Text_IO, Wide_Text_IO, and Stream_IO
```

by:

```
type File_Mode is (In_File, Out_File, Append_File);
--  for Sequential_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Stream_IO
```

Replace paragraph 13:

Several file management operations are common to Sequential_IO, Direct_IO, Text_IO, and Wide_Text_IO. These operations are described in subclause A.8.2 for sequential and direct files. Any additional effects concerning text input-output are described in subclause A.10.2.

by:

Several file management operations are common to Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. These operations are described in subclause A.8.2 for sequential and direct files. Any additional effects concerning text input-output are described in subclause A.10.2.

Replace paragraph 15:

18 Each instantiation of the generic packages Sequential_IO and Direct_IO declares a different type File_Type. In the case of Text_IO, Wide_Text_IO, and Streams.Stream_IO, the corresponding type File_Type is unique.

by:

18 Each instantiation of the generic packages Sequential_IO and Direct_IO declares a different type File_Type. In the case of Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Streams.Stream_IO, the corresponding type File_Type is unique.

A.8 Sequential and Direct Files

Replace paragraph 1:

Two kinds of access to external files are defined in this subclause: *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*. Access to stream files is described in A.12.1.

by:

Two kinds of access to external files are defined in this subclause: *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*. Access to *stream file*s is described in A.12.1.

A.8.1 The Generic Package Sequential_IO

Insert after paragraph 16:

```
private
   ... -- not specified by the language
end Ada.Sequential_IO;
```

the new paragraph:

The type File_Type needs finalization (see 7.6) in every instantiation of Sequential_IO.

A.8.2 File Management

Replace paragraph 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 defined.

by:

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, stream, 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 defined.

Replace paragraph 16:

Resets the given file so that reading from its elements can be restarted from the beginning of the file (for modes In_File and Inout_File), and so that writing to its elements can be restarted at the beginning of the file (for modes Out_File and Inout_File) or after the last element of the file (for mode Append_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. In addition, for sequential files, if the given file has mode Out_File or Append_File when Reset is called, the last element written since the most recent open or reset is the last element that can be read from the file. If no elements have been written and the file mode is Out_File, the reset file is empty. If no elements have been written and the file mode is Append_File, then the reset file is unchanged.

by:

Resets the given file so that reading from its elements can be restarted from the beginning of the external file (for modes In_File and Inout_File), and so that writing to its elements can be restarted at the beginning of the external file (for modes Out_File and Inout_File) or after the last element of the external file (for mode Append_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. In addition, for sequential files, if the given file has mode Out_File or Append_File when Reset is called, the last element written since the most recent open or reset is the last element that can be read from the external file. If no elements have been written and the file mode is Out_File, the reset file is empty. If no elements have been written and the file mode is Append_File, then the reset file is unchanged.

Replace paragraph 22:

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 external 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.

by:

Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an Open operation).

A.8.4 The Generic Package Direct_IO

Insert after paragraph 19:

```
private
   ... -- not specified by the language
end Ada.Direct_IO;
```

the new paragraph:

The type File_Type needs finalization (see 7.6) in every instantiation of Direct_IO.

A.10.1 The Package Text_IO

Insert after paragraph 48:

```
procedure Put(File : in  File_Type; Item : in String);
procedure Put(Item : in  String);
```

the new paragraphs:

```
function Get_Line(File : in  File_Type) return String;
function Get_Line return String;
```

Insert after paragraph 85:

```
   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
   ... -- not specified by the language
end Ada.Text_IO;
```

the new paragraph:

The type File_Type needs finalization (see 7.6).

A.10.6 Get and Put Procedures

In paragraph 5 replace:

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. Get procedures for numeric or enumeration types start by skipping leading blanks, where a *blank* is 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.

by:

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* is 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.

A.10.7 Input-Output of Characters and Strings

Replace paragraph 13:

For an item of type String the following procedures are provided:

by:

For an item of type String the following subprograms are provided:

Insert after paragraph 17:

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).

the new paragraphs:

```
function Get_Line(File : in  File_Type) return String;
function Get_Line return String;
```

Returns a result string constructed by reading successive characters from the specified input file, and assigning them to successive characters of the result string. The result string has a lower bound of 1 and an upper bound of the number of characters read. Reading stops when the end of the line is met; Skip_Line is then (in effect) called with a spacing of 1.

Constraint_Error is raised if the length of the line exceeds Positive'Last; in this case, the line number and page number are unchanged, and the column number is unspecified but no less than it was before the call. The exception End_Error is propagated if an attempt is made to skip a file terminator.

A.10.11 Input-Output for Bounded Strings

Insert new clause:

The package Text_IO.Bounded_IO provides input-output in human-readable form for Bounded_Strings.

Static Semantics

The generic library package Text_IO.Bounded_IO has the following declaration:

```
with Ada.Strings.Bounded;
generic
   with package Bounded is
                     new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
package Ada.Text_IO.Bounded_IO is

   procedure Put
      (File : in File_Type;
       Item : in Bounded.Bounded_String);

   procedure Put
      (Item : in Bounded.Bounded_String);

   procedure Put_Line
      (File : in File_Type;
       Item : in Bounded.Bounded_String);

   procedure Put_Line
      (Item : in Bounded.Bounded_String);
```

```
      function Get_Line
         (File : in File_Type)
         return Bounded.Bounded_String;

      function Get_Line
         return Bounded.Bounded_String;

      procedure Get_Line
         (File : in File_Type; Item : out Bounded.Bounded_String);

      procedure Get_Line
         (Item : out Bounded.Bounded_String);

   end Ada.Text_IO.Bounded_IO;
```

For an item of type Bounded_String, the following subprograms are provided:

```
   procedure Put
      (File : in File_Type;
       Item : in Bounded.Bounded_String);
```

Equivalent to Text_IO.Put (File, Bounded.To_String(Item));

```
   procedure Put
      (Item : in Bounded.Bounded_String);
```

Equivalent to Text_IO.Put (Bounded.To_String(Item));

```
   procedure Put_Line
      (File : in File_Type;
       Item : in Bounded.Bounded_String);
```

Equivalent to Text_IO.Put_Line (File, Bounded.To_String(Item));

```
   procedure Put_Line
      (Item : in Bounded.Bounded_String);
```

Equivalent to Text_IO.Put_Line (Bounded.To_String(Item));

```
   function Get_Line
      (File : in File_Type)
      return Bounded.Bounded_String;
```

Returns Bounded.To_Bounded_String(Text_IO.Get_Line(File));

```
   function Get_Line
      return Bounded.Bounded_String;
```

Returns Bounded.To_Bounded_String(Text_IO.Get_Line);

```
   procedure Get_Line
      (File : in File_Type; Item : out Bounded.Bounded_String);
```

Equivalent to Item := Get_Line (File);

```
   procedure Get_Line
      (Item : out Bounded.Bounded_String);
```

Equivalent to Item := Get_Line;

A.10.12 Input-Output for Unbounded Strings

Insert new clause:

The package Text_IO.Unbounded_IO provides input-output in human-readable form for Unbounded_Strings.

Static Semantics

The library package Text_IO.Unbounded_IO has the following declaration:

```
   with Ada.Strings.Unbounded;
```

```
package Ada.Text_IO.Unbounded_IO is

   procedure Put
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);

   procedure Put
      (Item : in Strings.Unbounded.Unbounded_String);

   procedure Put_Line
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);

   procedure Put_Line
      (Item : in Strings.Unbounded.Unbounded_String);

   function Get_Line
      (File : in File_Type)
      return Strings.Unbounded.Unbounded_String;

   function Get_Line
      return Strings.Unbounded.Unbounded_String;

   procedure Get_Line
      (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);

   procedure Get_Line
      (Item : out Strings.Unbounded.Unbounded_String);

end Ada.Text_IO.Unbounded_IO;
```

For an item of type Unbounded_String, the following subprograms are provided:

```
procedure Put
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

Equivalent to Text_IO.Put (File, Strings.Unbounded.To_String(Item));

```
procedure Put
   (Item : in Strings.Unbounded.Unbounded_String);
```

Equivalent to Text_IO.Put (Strings.Unbounded.To_String(Item));

```
procedure Put_Line
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

Equivalent to Text_IO.Put_Line (File, Strings.Unbounded.To_String(Item));

```
procedure Put_Line
   (Item : in Strings.Unbounded.Unbounded_String);
```

Equivalent to Text_IO.Put_Line (Strings.Unbounded.To_String(Item));

```
function Get_Line
   (File : in File_Type)
   return Strings.Unbounded.Unbounded_String;
```

Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line(File));

```
function Get_Line
   return Strings.Unbounded.Unbounded_String;
```

Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line);

```
procedure Get_Line
   (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

Equivalent to Item := Get_Line (File);

```
procedure Get_Line
   (Item : out Strings.Unbounded.Unbounded_String);
```

Equivalent to Item := Get_Line;

A.11 Wide Text Input-Output and Wide Wide Text Input-Output

Replace the title:

Wide Text Input-Output

by:

Wide Text Input-Output and Wide Wide Text Input-Output

Replace paragraph 1:

The package Wide_Text_IO provides facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of wide characters grouped into lines, and as a sequence of lines grouped into pages.

by:

The packages Wide_Text_IO and Wide_Wide_Text_IO provide facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of wide characters (or wide wide characters) grouped into lines, and as a sequence of lines grouped into pages.

Replace paragraph 2:

The specification of package Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line procedure, any occurrence of Character is replaced by Wide_Character, and any occurrence of String is replaced by Wide_String.

by:

The specification of package Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Character, and any occurrence of String is replaced by Wide_String. Nongeneric equivalents of Wide_Text_IO.Integer_IO and Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Text_IO, Ada.Long_Integer_Wide_Text_IO, Ada.Float_Wide_Text_IO, Ada.Long_Float_Wide_Text_IO.

Replace paragraph 3:

Nongeneric equivalents of Wide_Text_IO.Integer_IO and Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Text_IO, Ada.Long_Integer_Wide_Text_IO, Ada.Float_Wide_Text_IO, Ada.Long_Float_Wide_Text_IO.

by:

The specification of package Wide_Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Wide_Character, and any occurrence of String is replaced by Wide_Wide_String. Nongeneric equivalents of Wide_Wide_Text_IO.Integer_IO and Wide_Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Wide_Text_IO, Ada.Long_Integer_Wide_Wide_Text_IO, Ada.Float_Wide_Wide_Text_IO, Ada.Long_Float_Wide_Wide_Text_IO.

The specification of package Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Bounded. The specification of package Wide_Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Wide_Bounded.

The specification of package Wide_Text_IO.Wide_Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Wide_Unbounded_String, and any occurrence of package Unbounded is replaced by Wide_Unbounded. The specification of package

Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Wide_Wide_Unbounded_String, and any occurrence of package Unbounded is replaced by Wide_Wide_Unbounded.

A.12 Stream Input-Output

Replace paragraph 1:

The packages Streams.Stream_IO, Text_IO.Text_Streams, and Wide_Text_IO.Text_Streams provide stream-oriented operations on files.

by:

The packages Streams.Stream_IO, Text_IO.Text_Streams, Wide_Text_IO.Text_Streams, and Wide_Wide_Text_IO.Text_Streams provide stream-oriented operations on files.

A.12.1 The Package Streams.Stream_IO

Insert after paragraph 27:

```
private
   ... -- not specified by the language
end Ada.Streams.Stream_IO;
```

the new paragraph:

The type File_Type needs finalization (see 7.6).

Replace paragraph 28:

The subprograms Create, Open, Close, Delete, Reset, Mode, Name, Form, Is_Open, and End_of_File have the same effect as the corresponding subprograms in Sequential_IO (see A.8.2).

by:

The subprograms given in subclause A.8.2 for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, and Is_Open) are available for stream files.

The End_Of_File function:

- Propagates Mode_Error if the mode of the file is not In_File;
- If positioning is supported for the given external file, the function returns True if the current index exceeds the size of the external file; otherwise it returns False;
- If positioning is not supported for the given external file, the function returns True if no more elements can be read from the given file; otherwise it returns False.

Replace paragraph 28.1:

The Set_Mode procedure changes the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

by:

The Set_Mode procedure sets the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

Replace paragraph 30:

The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index.

by:

The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index. For a file that supports positioning, Read without a Positive_Count parameter starts reading at the current index, and Write without a Positive_Count parameter starts writing at the current index.

A.12.4 The Package Wide_Wide_Text_IO.Text_Streams

Insert new clause:

The package Wide_Wide_Text_IO.Text_Streams provides a function for treating a wide wide text file as a stream.

Static Semantics

The library package Wide_Wide_Text_IO.Text_Streams has the following declaration:

```
with Ada.Streams;
package Ada.Wide_Wide_Text_IO.Text_Streams is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;

   function Stream (File : in File_Type) return Stream_Access;
end Ada.Wide_Wide_Text_IO.Text_Streams;
```

The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

A.16 The Package Directories

Insert new clause:

The package Directories provides operations for manipulating files and directories, and their names.

Static Semantics

The library package Directories has the following declaration:

```
with Ada.IO_Exceptions;
with Ada.Calendar;
package Ada.Directories is

   -- Directory and file operations:

   function Current_Directory return String;

   procedure Set_Directory (Directory : in String);

   procedure Create_Directory (New_Directory : in String;
                               Form : in String := "");

   procedure Delete_Directory (Directory : in String);

   procedure Create_Path (New_Directory : in String;
                          Form : in String := "");

   procedure Delete_Tree (Directory : in String);

   procedure Delete_File (Name : in String);

   procedure Rename (Old_Name, New_Name : in String);

   procedure Copy_File (Source_Name, Target_Name : in String;
                        Form : in String := "");

   -- File and directory name operations:
```

```
function Full_Name (Name : in String) return String;

function Simple_Name (Name : in String) return String;

function Containing_Directory (Name : in String) return String;

function Extension (Name : in String) return String;

function Base_Name (Name : in String) return String;

function Compose (Containing_Directory : in String := "";
                  Name : in String;
                  Extension : in String := "") return String;

-- File and directory queries:

type File_Kind is (Directory, Ordinary_File, Special_File);

type File_Size is range 0 .. implementation-defined;

function Exists (Name : in String) return Boolean;

function Kind (Name : in String) return File_Kind;

function Size (Name : in String) return File_Size;

function Modification_Time (Name : in String) return Ada.Calendar.Time;

-- Directory searching:

type Directory_Entry_Type is limited private;

type Filter_Type is array (File_Kind) of Boolean;

type Search_Type is limited private;

procedure Start_Search (Search    : in out Search_Type;
                        Directory : in String;
                        Pattern   : in String;
                        Filter    : in Filter_Type := (others => True));

procedure End_Search (Search : in out Search_Type);

function More_Entries (Search : in Search_Type) return Boolean;

procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);

procedure Search (
    Directory : in String;
    Pattern   : in String;
    Filter    : in Filter_Type := (others => True);
    Process   : not null access procedure (
        Directory_Entry : in Directory_Entry_Type));

-- Operations on Directory Entries:

function Simple_Name (Directory_Entry : in Directory_Entry_Type)
    return String;

function Full_Name (Directory_Entry : in Directory_Entry_Type)
    return String;

function Kind (Directory_Entry : in Directory_Entry_Type)
    return File_Kind;
```

```
        function Size (Directory_Entry : in Directory_Entry_Type)
            return File_Size;

        function Modification_Time (Directory_Entry : in Directory_Entry_Type)
            return Ada.Calendar.Time;

        Status_Error : exception renames Ada.IO_Exceptions.Status_Error;
        Name_Error   : exception renames Ada.IO_Exceptions.Name_Error;
        Use_Error    : exception renames Ada.IO_Exceptions.Use_Error;
        Device_Error : exception renames Ada.IO_Exceptions.Device_Error;

    private
        -- Not specified by the language.
    end Ada.Directories;
```

External files may be classified as directories, special files, or ordinary files. A *directory* is an external file that is a container for files on the target system. A *special file* is an external file that cannot be created or read by a predefined Ada input-output package. External files that are not special files or directories are called *ordinary files*.

A *file name* is a string identifying an external file. Similarly, a *directory name* is a string identifying a directory. The interpretation of file names and directory names is implementation-defined.

The *full name* of an external file is a full specification of the name of the file. If the external environment allows alternative specifications of the name (for example, abbreviations), the full name should not use such alternatives. A full name typically will include the names of all of the directories that contain the item. The *simple name* of an external file is the name of the item, not including any containing directory names. Unless otherwise specified, a file name or directory name parameter in a call to a predefined Ada input-output subprogram can be a full name, a simple name, or any other form of name supported by the implementation.

The *default directory* is the directory that is used if a directory or file name is not a full name (that is, when the name does not fully identify all of the containing directories).

A *directory entry* is a single item in a directory, identifying a single external file (including directories and special files).

For each function that returns a string, the lower bound of the returned value is 1.

The following file and directory operations are provided:

```
function Current_Directory return String;
```

Returns the full directory name for the current default directory. The name returned shall be suitable for a future call to Set_Directory. The exception Use_Error is propagated if a default directory is not supported by the external environment.

```
procedure Set_Directory (Directory : in String);
```

Sets the current default directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support making Directory (in the absence of Name_Error) a default directory.

```
procedure Create_Directory (New_Directory : in String;
                            Form : in String := "");
```

Creates a directory with name New_Directory. The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of a directory. The exception Use_Error is propagated if the external environment does not support the creation of a directory with the given name (in the absence of Name_Error) and form.

```
procedure Delete_Directory (Directory : in String);
```

Deletes an existing empty directory with name Directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory (or some portion of its contents) with the given name (in the absence of Name_Error).

```
procedure Create_Path (New_Directory : in String;
                       Form : in String := "");
```

Creates zero or more directories with name New_Directory. Each non-existent directory named by New_Directory is created. For example, on a typical Unix system, Create_Path ("/usr/me/my"); would create directory "me" in directory "usr", then create directory "my" in directory "me". The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of any directory. The exception Use_Error is propagated if the external environment does not support the creation of any directories with the given name (in the absence of Name_Error) and form.

```
procedure Delete_Tree (Directory : in String);
```

Deletes an existing directory with name Directory. The directory and all of its contents (possibly including other directories) are deleted. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory or some portion of its contents with the given name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the contents of the directory is deleted.

```
procedure Delete_File (Name : in String);
```

Deletes an existing ordinary or special file with name Name. The exception Name_Error is propagated if the string given as Name does not identify an existing ordinary or special external file. The exception Use_Error is propagated if the external environment does not support the deletion of the file with the given name (in the absence of Name_Error).

```
procedure Rename (Old_Name, New_Name : in String);
```

Renames an existing external file (including directories) with name Old_Name to New_Name. The exception Name_Error is propagated if the string given as Old_Name does not identify an existing external file. The exception Use_Error is propagated if the external environment does not support the renaming of the file with the given name (in the absence of Name_Error). In particular, Use_Error is propagated if a file or directory already exists with name New_Name.

```
procedure Copy_File (Source_Name, Target_Name : in String;
                     Form : in String);
```

Copies the contents of the existing external file with name Source_Name to an external file with name Target_Name. The resulting external file is a duplicate of the source external file. The Form parameter can be used to give system-dependent characteristics of the resulting external file; the interpretation of the Form parameter is implementation-defined. Exception Name_Error is propagated if the string given as Source_Name does not identify an existing external ordinary or special file, or if the string given as Target_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external environment does not support creating the file with the name given by Target_Name and form given by Form, or copying of the file with the name given by Source_Name (in the absence of Name_Error).

The following file and directory name operations are provided:

```
function Full_Name (Name : in String) return String;
```

Returns the full name corresponding to the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Simple_Name (Name : in String) return String;
```

Returns the simple name portion of the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Containing_Directory (Name : in String) return String;
```

Returns the name of the containing directory of the external file (including directories) identified by Name. (If more than one directory can contain Name, the directory name returned is implementation-defined.) The

exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if the external file does not have a containing directory.

```
function Extension (Name : in String) return String;
```

Returns the extension name corresponding to Name. The extension name is a portion of a simple name (not including any separator characters), typically used to identify the file class. If the external environment does not have extension names, then the null string is returned. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file.

```
function Base_Name (Name : in String) return String;
```

Returns the base name corresponding to Name. The base name is the remainder of a simple name after removing any extension and extension separators. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Compose (Containing_Directory : in String := "";
                  Name : in String;
                  Extension : in String := "") return String;
```

Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise Name is interpreted as a base name. The exception Name_Error is propagated if the string given as Containing_Directory is not null and does not allow the identification of a directory, or if the string given as Extension is not null and is not a possible extension, or if the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is non-null).

The following file and directory queries and types are provided:

```
type File_Kind is (Directory, Ordinary_File, Special_File);
```

The type File_Kind represents the kind of file represented by an external file or directory.

```
type File_Size is range 0 .. implementation-defined;
```

The type File_Size represents the size of an external file.

```
function Exists (Name : in String) return Boolean;
```

Returns True if an external file represented by Name exists, and False otherwise. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Kind (Name : in String) return File_Kind;
```

Returns the kind of external file represented by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file.

```
function Size (Name : in String) return File_Size;
```

Returns the size of the external file represented by Name. The size of an external file is the number of stream elements contained in the file. If the external file is not an ordinary file, the result is implementation-defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

```
function Modification_Time (Name : in String) return Ada.Calendar.Time;
```

Returns the time that the external file represented by Name was most recently modified. If the external file is not an ordinary file, the result is implementation-defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file with the name given by Name (in the absence of Name_Error).

The following directory searching operations and types are provided:

```
type Directory_Entry_Type is limited private;
```

The type Directory_Entry_Type represents a single item in a directory. These items can only be created by the Get_Next_Entry procedure in this package. Information about the item can be obtained from the functions

declared in this package. A default-initialized object of this type is invalid; objects returned from Get_Next_Entry are valid.

```
type Filter_Type is array (File_Kind) of Boolean;
```

The type Filter_Type specifies which directory entries are provided from a search operation. If the Directory component is True, directory entries representing directories are provided. If the Ordinary_File component is True, directory entries representing ordinary files are provided. If the Special_File component is True, directory entries representing special files are provided.

```
type Search_Type is limited private;
```

The type Search_Type contains the state of a directory search. A default-initialized Search_Type object has no entries available (function More_Entries returns False). Type Search_Type needs finalization (see 7.6).

```
procedure Start_Search (Search    : in out Search_Type;
                        Directory : in String;
                        Pattern   : in String;
                        Filter    : in Filter_Type := (others => True));
```

Starts a search in the directory named by Directory for entries matching Pattern. Pattern represents a pattern for matching file names. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation-defined. Only items that match Filter will be returned. After a successful call on Start_Search, the object Search may have entries available, but it may have no entries available if no files or directories match Pattern and Filter. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error). When Start_Search propagates Name_Error or Use_Error, the object Search will have no entries available.

```
procedure End_Search (Search : in out Search_Type);
```

Ends the search represented by Search. After a successful call on End_Search, the object Search will have no entries available.

```
function More_Entries (Search : in Search_Type) return Boolean;
```

Returns True if more entries are available to be returned by a call to Get_Next_Entry for the specified search object, and False otherwise.

```
procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);
```

Returns the next Directory_Entry for the search described by Search that matches the pattern and filter. If no further matches are available, Status_Error is raised. It is implementation-defined as to whether the results returned by this routine are altered if the contents of the directory are altered while the Search object is valid (for example, by another program). The exception Use_Error is propagated if the external environment does not support continued searching of the directory represented by Search.

```
procedure Search (
    Directory : in String;
    Pattern   : in String;
    Filter    : in Filter_Type := (others => True);
    Process   : not null access procedure (
        Directory_Entry : in Directory_Entry_Type));
```

Searches in the directory named by Directory for entries matching Pattern. The subprogram designated by Process is called with each matching entry in turn. Pattern represents a pattern for matching file names. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation-defined. Only items that match Filter will be returned. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error).

```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

Returns the simple external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid.

```
function Full_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

Returns the full external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid.

```
function Kind (Directory_Entry : in Directory_Entry_Type)
     return File_Kind;
```

Returns the kind of external file represented by Directory_Entry. The exception Status_Error is propagated if Directory_Entry is invalid.

```
function Size (Directory_Entry : in Directory_Entry_Type)
     return File_Size;
```

Returns the size of the external file represented by Directory_Entry. The size of an external file is the number of stream elements contained in the file. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
     return Ada.Calendar.Time;
```

Returns the time that the external file represented by Directory_Entry was most recently modified. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation-defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file represented by Directory_Entry.

Implementation Requirements

For Copy_File, if Source_Name identifies an existing external ordinary file created by a predefined Ada input-output package, and Target_Name and Form can be used in the Create operation of that input-output package with mode Out_File without raising an exception, then Copy_File shall not propagate Use_Error.

Implementation Advice

If other information about a file (such as the owner or creation date) is available in a directory entry, the implementation should provide functions in a child package Directories.Information to retrieve it.

Start_Search and Search should raise Use_Error if Pattern is malformed, but not if it could represent a file in the directory but does not actually do so.

Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file.

NOTES

37 The operations Containing_Directory, Full_Name, Simple_Name, Base_Name, Extension, and Compose operate on file names, not external files. The files identified by these operations do not need to exist. Name_Error is raised only if the file name is malformed and cannot possibly identify a file. Of these operations, only the result of Full_Name depends on the current default directory; the result of the others depends only on their parameters.

38 Using access types, values of Search_Type and Directory_Entry_Type can be saved and queried later. However, another task or application can modify or delete the file represented by a Directory_Entry_Type value or the directory represented by a Search_Type value; such a value can only give the information valid at the time it is created. Therefore, long-term storage of these values is not recommended.

39 If the target system does not support directories inside of directories, then Kind will never return Directory and Containing_Directory will always raise Use_Error.

40 If the target system does not support creation or deletion of directories, then Create_Directory, Create_Path, Delete_Directory, and Delete_Tree will always propagate Use_Error.

41 To move a file or directory to a different location, use Rename. Most target systems will allow renaming of files from one directory to another. If the target file or directory might already exist, it should be deleted first.

A.17 The Package Environment_Variables

Insert new clause:

The package Environment_Variables allows a program to read or modify environment variables. Environment variables are name-value pairs, where both the name and value are strings. The definition of what constitutes an *environment variable*, and the meaning of the name and value, are implementation defined.

Static Semantics

The library package Environment_Variables has the following declaration:

```
package Ada.Environment_Variables is
   pragma Preelaborate(Environment_Variables);

   function Value (Name : in String) return String;

   function Exists (Name : in String) return Boolean;

   procedure Set (Name : in String; Value : in String);

   procedure Clear (Name : in String);
   procedure Clear;

   procedure Iterate (
       Process : not null access procedure (Name, Value : in String));

end Ada.Environment_Variables;
```

`function Value (Name : in String) return String;`

If the external execution environment supports environment variables, then Value returns the value of the environment variable with the given name. If no environment variable with the given name exists, then Constraint_Error is propagated. If the execution environment does not support environment variables, then Program_Error is propagated.

`function Exists (Name : in String) return Boolean;`

If the external execution environment supports environment variables and an environment variable with the given name currently exists, then Exists returns True; otherwise it returns False.

`procedure Set (Name : in String; Value : in String);`

If the external execution environment supports environment variables, then Set first clears any existing environment variable with the given name, and then defines a single new environment variable with the given name and value. Otherwise Program_Error is propagated.

If implementation-defined circumstances prohibit the definition of an environment variable with the given name and value, then Constraint_Error is propagated.

It is implementation defined whether there exist values for which the call Set(Name, Value) has the same effect as Clear (Name).

`procedure Clear (Name : in String);`

If the external execution environment supports environment variables, then Clear deletes all existing environment variable with the given name. Otherwise Program_Error is propagated.

`procedure Clear;`

If the external execution environment supports environment variables, then Clear deletes all existing environment variables. Otherwise Program_Error is propagated.

```
procedure Iterate (
     Process : not null access procedure (Name, Value : in String));
```

If the external execution environment supports environment variables, then Iterate calls the subprogram designated by Process for each existing environment variable, passing the name and value of that environment variable. Otherwise Program_Error is propagated.

If several environment variables exist that have the same name, Process is called once for each such variable.

Bounded (Run-Time) Errors

It is a bounded error to call Value if more than one environment variable exists with the given name; the possible outcomes are that:

- one of the values is returned, and that same value is returned in subsequent calls in the absence of changes to the environment; or
- Program_Error is propagated.

Erroneous Execution

Making calls to the procedures Set or Clear concurrently with calls to any subprogram of package Environment_Variables, or to any instantiation of Iterate, results in erroneous execution.

Making calls to the procedures Set or Clear in the actual subprogram corresponding to the Process parameter of Iterate results in erroneous execution.

Documentation Requirements

An implementation shall document how the operations of this package behave if environment variables are changed by external mechanisms (for instance, calling operating system services).

Implementation Permissions

An implementation running on a system that does not support environment variables is permitted to define the operations of package Environment_Variables with the semantics corresponding to the case where the external execution environment does support environment variables. In this case, it shall provide a mechanism to initialize a nonempty set of environment variables prior to the execution of a partition.

Implementation Advice

If the execution environment supports subprocesses, the currently defined environment variables should be used to initialize the environment variables of a subprocess.

Changes to the environment variables made outside the control of this package should be reflected immediately in the effect of the operations of this package. Changes to the environment variables made using this package should be reflected immediately in the external execution environment. This package should not perform any buffering of the environment variables.

A.18 Containers

Insert new clause:

This clause presents the specifications of the package Containers and several child packages, which provide facilities for storing collections of elements.

A variety of sequence and associative containers are provided. Each container includes a *cursor* type. A cursor is a reference to an element within a container. Many operations on cursors are common to all of the containers. A cursor referencing an element in a container is considered to be overlapping with the container object itself.

Within this clause we provide Implementation Advice for the desired average or worst case time complexity of certain operations on a container. This advice is expressed using the Landau symbol $O(X)$. Presuming f is some function of a length parameter N and t(N) is the time the operation takes (on average or worst case, as specified) for the length N, a complexity of $O(f(N))$ means that there exists a finite A such that for any N, $t(N)/f(N) < A$.

If the advice suggests that the complexity should be less than $O(f(N))$, then for any arbitrarily small positive real D, there should exist a positive integer M such that for all $N > M$, $t(N)/f(N) < D$.

A.18.1 The Package Containers

Insert new clause:

The package Containers is the root of the containers subsystem.

Static Semantics

The library package Containers has the following declaration:

```
package Ada.Containers is
   pragma Pure(Containers);

   type Hash_Type is mod implementation-defined;

   type Count_Type is range 0 .. implementation-defined;

end Ada.Containers;
```

Hash_Type represents the range of the result of a hash function. Count_Type represents the (potential or actual) number of elements of a container.

Implementation Advice

Hash_Type'Modulus should be at least 2**32. Count_Type'Last should be at least 2**31–1.

A.18.2 The Package Containers.Vectors

Insert new clause:

The language-defined generic package Containers.Vectors provides private types Vector and Cursor, and a set of operations for each type. A vector container allows insertion and deletion at any position, but it is specifically optimized for insertion and deletion at the high end (the end with the higher index) of the container. A vector container also provides random access to its elements.

A vector container behaves conceptually as an array that expands as necessary as items are inserted. The *length* of a vector is the number of elements that the vector contains. The *capacity* of a vector is the maximum number of elements that can be inserted into the vector prior to it being automatically expanded.

Elements in a vector container can be referred to by an index value of a generic formal type. The first element of a vector always has its index value equal to the lower bound of the formal type.

A vector container may contain *empty elements*. Empty elements do not have a specified value.

Static Semantics

The generic library package Containers.Vectors has the following declaration:

```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors is
   pragma Preelaborate(Vectors);

   subtype Extended_Index is
      Index_Type'Base range
         Index_Type'First-1 ..
         Index_Type'Min (Index_Type'Base'Last - 1, Index_Type'Last) + 1;
   No_Index : constant Extended_Index := Extended_Index'First;

   type Vector is tagged private;
   pragma Preelaborable_Initialization(Vector);

   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
```

```
Empty_Vector : constant Vector;

No_Element : constant Cursor;

function "=" (Left, Right : Vector) return Boolean;

function To_Vector (Length : Count_Type) return Vector;

function To_Vector
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector;

function "&" (Left, Right : Vector) return Vector;

function "&" (Left  : Vector;
              Right : Element_Type) return Vector;

function "&" (Left  : Element_Type;
              Right : Vector) return Vector;

function "&" (Left, Right  : Element_Type) return Vector;

function Capacity (Container : Vector) return Count_Type;

procedure Reserve_Capacity (Container : in out Vector;
                            Capacity  : in     Count_Type);

function Length (Container : Vector) return Count_Type;

procedure Set_Length (Container : in out Vector;
                      Length    : in     Count_Type);

function Is_Empty (Container : Vector) return Boolean;

procedure Clear (Container : in out Vector);

function To_Cursor (Container : Vector;
                    Index     : Extended_Index) return Cursor;

function To_Index (Position  : Cursor) return Extended_Index;

function Element (Container : Vector;
                  Index     : Index_Type)
   return Element_Type;

function Element (Position : Cursor) return Element_Type;

procedure Replace_Element (Container : in out Vector;
                           Index     : in     Index_Type;
                           New_Item  : in     Element_Type);

procedure Replace_Element (Container : in out Vector;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);

procedure Query_Element
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type));

procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));

procedure Update_Element
  (Container : in out Vector;
   Index     : in     Index_Type;
```

```
   Process   : not null access procedure
                  (Element : in out Element_Type));

procedure Update_Element
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Element : in out Element_Type));

procedure Move (Target : in out Vector;
                Source : in out Vector);

procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Vector);

procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector);

procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector;
                  Position  :    out Cursor);

procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);

procedure Prepend (Container : in out Vector;
                   New_Item  : in     Vector);

procedure Prepend (Container : in out Vector;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);

procedure Append (Container : in out Vector;
                  New_Item  : in     Vector);

procedure Append (Container : in out Vector;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);

procedure Insert_Space (Container : in out Vector;
                        Before    : in     Extended_Index;
```

```
                        Count     : in     Count_Type := 1);

   procedure Insert_Space (Container : in out Vector;
                           Before    : in     Cursor;
                           Position  :    out Cursor;
                           Count     : in     Count_Type := 1);

   procedure Delete (Container : in out Vector;
                     Index     : in     Extended_Index;
                     Count     : in     Count_Type := 1);

   procedure Delete (Container : in out Vector;
                     Position  : in out Cursor;
                     Count     : in     Count_Type := 1);

   procedure Delete_First (Container : in out Vector;
                           Count     : in     Count_Type := 1);

   procedure Delete_Last (Container : in out Vector;
                          Count     : in     Count_Type := 1);

   procedure Reverse_Elements (Container : in out Vector);

   procedure Swap (Container : in out Vector;
                   I, J      : in     Index_Type);

   procedure Swap (Container : in out Vector;
                   I, J      : in     Cursor);

   function First_Index (Container : Vector) return Index_Type;

   function First (Container : Vector) return Cursor;

   function First_Element (Container : Vector)
      return Element_Type;

   function Last_Index (Container : Vector) return Extended_Index;

   function Last (Container : Vector) return Cursor;

   function Last_Element (Container : Vector)
      return Element_Type;

   function Next (Position : Cursor) return Cursor;

   procedure Next (Position : in out Cursor);

   function Previous (Position : Cursor) return Cursor;

   procedure Previous (Position : in out Cursor);

   function Find_Index (Container : Vector;
                        Item      : Element_Type;
                        Index     : Index_Type := Index_Type'First)
      return Extended_Index;

   function Find (Container : Vector;
                  Item      : Element_Type;
                  Position  : Cursor := No_Element)
      return Cursor;

   function Reverse_Find_Index (Container : Vector;
                                Item      : Element_Type;
                                Index     : Index_Type := Index_Type'Last)
      return Extended_Index;

   function Reverse_Find (Container : Vector;
```

```
                        Item      : Element_Type;
                        Position  : Cursor := No_Element)
      return Cursor;

   function Contains (Container : Vector;
                      Item      : Element_Type) return Boolean;


   function Has_Element (Position : Cursor) return Boolean;

   procedure  Iterate
     (Container : in Vector;
      Process   : not null access procedure (Position : in Cursor));

   procedure Reverse_Iterate
     (Container : in Vector;
      Process   : not null access procedure (Position : in Cursor));

   generic
      with function "<" (Left, Right : Element_Type)
         return Boolean is <>;
   package Generic_Sorting is

      function Is_Sorted (Container : Vector) return Boolean;

      procedure Sort (Container : in out Vector);

      procedure Merge (Target  : in out Vector;
                       Source  : in out Vector);

   end Generic_Sorting;

private

   ... -- not specified by the language

end Ada.Containers.Vectors;
```

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions defined to use it return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions defined to use it are unspecified.

The type Vector is used to represent vectors. The type Vector needs finalization (see 7.6).

Empty_Vector represents the empty vector object. It has a length of 0. If an object of type Vector is not otherwise initialized, it is initialized to the same value as Empty_Vector.

No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

No_Index represents a position that does not correspond to any element. The subtype Extended_Index includes the indices covered by Index_Type plus the value No_Index and, if it exists, the successor to the Index_Type'Last.

Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

A subprogram is said to *tamper with cursors* of a vector object *V* if:

- it inserts or deletes elements of *V*, that is, it calls the Insert, Insert_Space, Clear, Delete, or Set_Length procedures with *V* as a parameter; or
- it finalizes *V*; or
- it calls the Move procedure with *V* as a parameter.

A subprogram is said to *tamper with elements* of a vector object *V* if:

- it tampers with cursors of *V*; or
- it replaces one or more elements of *V*, that is, it calls the Replace_Element, Reverse_Elements, or Swap procedures or the Sort or Merge procedures of an instance of Generic_Sorting with *V* as a parameter.

```
function "=" (Left, Right : Vector) return Boolean;
```

If Left and Right denote the same vector object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise it returns True. Any exception raised during evaluation of element equality is propagated.

```
function To_Vector (Length : Count_Type) return Vector;
```

Returns a vector with a length of Length, filled with empty elements.

```
function To_Vector
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector;
```

Returns a vector with a length of Length, filled with elements initialized to the value New_Item.

```
function "&" (Left, Right : Vector) return Vector;
```

Returns a vector comprising the elements of Left followed by the elements of Right.

```
function "&" (Left  : Vector;
              Right : Element_Type) return Vector;
```

Returns a vector comprising the elements of Left followed by the element Right.

```
function "&" (Left  : Element_Type;
              Right : Vector) return Vector;
```

Returns a vector comprising the element Left followed by the elements of Right.

```
function "&" (Left, Right  : Element_Type) return Vector;
```

Returns a vector comprising the element Left followed by the element Right.

```
function Capacity (Container : Vector) return Count_Type;
```

Returns the capacity of Container.

```
procedure Reserve_Capacity (Container : in out Vector;
                            Capacity  : in     Count_Type);
```

Reserve_Capacity allocates new internal data structures such that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then copies the elements into the new data structures and deallocates the old data structures. Any exception raised during allocation is propagated and Container is not modified.

```
function Length (Container : Vector) return Count_Type;
```

Returns the number of elements in Container.

```
procedure Set_Length (Container : in out Vector;
                      Length    : in     Count_Type);
```

If Length is larger than the capacity of Container, Set_Length calls Reserve_Capacity (Container, Length), then sets the length of the Container to Length. If Length is greater than the original length of Container, empty elements are added to Container; otherwise elements are removed from Container.

```
function Is_Empty (Container : Vector) return Boolean;
```

Equivalent to Length (Container) = 0.

```
procedure Clear (Container : in out Vector);
```

Removes all the elements from Container. The capacity of Container does not change.

```
function To_Cursor (Container : Vector;
                    Index     : Extended_Index) return Cursor;
```

If Index is not in the range First_Index (Container) .. Last_Index (Container), then No_Element is returned. Otherwise, a cursor designating the element at position Index in Container is returned.

```
function To_Index (Position  : Cursor) return Extended_Index;
```

If Position is No_Element, No_Index is returned. Otherwise, the index (within its containing vector) of the element designated by Position is returned.

```
function Element (Container : Vector;
                  Index     : Index_Type)
   return Element_Type;
```

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Element returns the element at position Index.

```
function Element (Position  : Cursor) return Element_Type;
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

```
procedure Replace_Element (Container : in out Vector;
                           Index     : in     Index_Type;
                           New_Item  : in     Element_Type);
```

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element at position Index. Any exception raised during the assignment is propagated. The element at position Index is not an empty element after successful call to Replace_Element.

```
procedure Replace_Element (Container : in out Vector;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns New_Item to the element designated by Position. Any exception raised during the assignment is propagated. The element at Position is not an empty element after successful call to Replace_Element.

```
procedure Query_Element
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type));
```

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element at position Index as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

```
procedure Update_Element
  (Container : in out Vector;
   Index     : in     Index_Type;
   Process   : not null access procedure (Element : in out Element_Type));
```

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Update_Element calls Process.**all** with the element at position Index as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

The element at position Index is not an empty element after successful completion of this operation.

```
procedure Update_Element
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure (Element : in out Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

The element designated by Position is not an empty element after successful completion of this operation.

```
procedure Move (Target : in out Vector;
                Source : in out Vector);
```

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order. The length of Source is 0 after a successful call to Move.

```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Vector);
```

If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Length(New_Item) is 0, then Insert does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Length (New_Item); if the value of Last appropriate for length *NL* would be greater than Index_Type'Last then Constraint_Error is propagated.

If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Length(New_Item) positions, and then copies the elements of New_Item to the positions starting at Before. Any exception raised during the copying is propagated.

```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, if Length(New_Item) is 0, then Insert does nothing. If Before is No_Element, then the call is equivalent to Insert (Container, Last_Index (Container) + 1, New_Item); otherwise the call is equivalent to Insert (Container, To_Index (Before), New_Item);

```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Vector;
```

```
                  Position  :    out Cursor);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert (Container, *T*, New_Item) is called, and then Position is set to To_Cursor (Container, *T*).

```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, Before, To_Vector (New_Item, Count));

```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, Before, To_Vector (New_Item, Count));

```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, Before, To_Vector (New_Item, Count), Position);

```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```

If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, then Insert does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last then Constraint_Error is propagated.

If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts elements that are initialized by default (see 3.3.1) in the positions starting at Before.

```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Vector;
                   Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, First_Index (Container), New_Item).

```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, First_Index (Container), New_Item, Count).

```
procedure Append (Container : in out Vector;
                  New_Item  : in     Vector);
```

Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item).

```
procedure Append (Container : in out Vector;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, Count).

```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1);
```

If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, then Insert_Space does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last then Constraint_Error is propagated.

If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert_Space slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts empty elements in the positions starting at Before.

```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert_Space (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

```
procedure Delete (Container : in out Vector;
                  Index     : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```

If Index is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, Delete has no effect. Otherwise Delete slides the elements (if any) starting at position Index + Count down to Index. Any exception raised during element assignment is propagated.

```
procedure Delete (Container : in out Vector;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);
```

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete (Container, To_Index (Position), Count) is called, and then Position is set to No_Element.

```
procedure Delete_First (Container : in out Vector;
                        Count     : in     Count_Type := 1);
```

Equivalent to Delete (Container, First_Index (Container), Count).

```
procedure Delete_Last (Container : in out Vector;
                       Count     : in     Count_Type := 1);
```

If Length (Container) <= Count then Delete_Last is equivalent to Clear (Container). Otherwise it is equivalent to Delete (Container, Index_Type'Val(Index_Type'Pos(Last_Index (Container)) – Count + 1), Count).

```
procedure Reverse_Elements (Container : in out Vector);
```

Reorders the elements of Container in reverse order.

```
procedure Swap (Container : in out Vector;
                I, J      : in     Index_Type);
```

If either I or J is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Swap exchanges the values of the elements at positions I and J.

```
procedure Swap (Container : in out Vector;
                I, J      : in     Cursor);
```

If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

```
function First_Index (Container : Vector) return Index_Type;
```

Returns the value Index_Type'First.

```
function First (Container : Vector) return Cursor;
```

If Container is empty, First returns No_Element. Otherwise, it returns a cursor that designates the first element in Container.

```
function First_Element (Container : Vector) return Element_Type;
```

Equivalent to Element (Container, First_Index (Container)).

```
function Last_Index (Container : Vector) return Extended_Index;
```

If Container is empty, Last_Index returns No_Index. Otherwise, it returns the position of the last element in Container.

```
function Last (Container : Vector) return Cursor;
```

If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last element in Container.

```
function Last_Element (Container : Vector) return Element_Type;
```

Equivalent to Element (Container, Last_Index (Container)).

```
function Next (Position : Cursor) return Cursor;
```

If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) + 1 in the same vector as Position.

```
procedure Next (Position : in out Cursor);
```

Equivalent to Position := Next (Position).

```
function Previous (Position : Cursor) return Cursor;
```

If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) – 1 in the same vector as Position.

```
procedure Previous (Position : in out Cursor);
```

Equivalent to Position := Previous (Position).

```
function Find_Index (Container : Vector;
                     Item      : Element_Type;
                     Index     : Index_Type := Index_Type'First)
   return Extended_Index;
```

Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index and proceeds towards Last_Index (Container). If no equal element is found, then Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

```
function Find (Container : Vector;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor;
```

If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the first element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the last element of Container. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Reverse_Find_Index (Container : Vector;
                             Item      : Element_Type;
                             Index     : Index_Type := Index_Type'Last)
   return Extended_Index;
```

Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index or, if Index is greater than Last_Index (Container), at position Last_Index

(Container). It proceeds towards First_Index (Container). If no equal element is found, then Reverse_Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

```
function Reverse_Find (Container : Vector;
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;
```

If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise Reverse_Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the last element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the first element of Container. If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Contains (Container : Vector;
                   Item      : Element_Type) return Boolean;
```

Equivalent to Has_Element (Find (Container, Item)).

```
function Has_Element (Position : Cursor) return Boolean;
```

Returns True if Position designates an element, and returns False otherwise.

```
procedure Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));
```

Invokes Process.**all** with a cursor that designates each element in Container, in index order. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process is propagated.

```
procedure Reverse_Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));
```

Iterates over the elements in Container as per Iterate, except that elements are traversed in reverse index order.

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

```
function Is_Sorted (Container : Vector) return Boolean;
```

Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

```
procedure Sort (Container : in out Vector);
```

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

```
procedure Merge (Target  : in out Vector;
                 Source  : in out Vector);
```

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

Bounded (Run-Time) Errors

Reading the value of an empty element by calling Element, Query_Element, Update_Element, Swap, Is_Sorted, Sort, Merge, "=", Find, or Reverse_Find is a bounded error. The implementation may treat the element as having any normal value (see 13.9.1) of the element type, or raise Constraint_Error or Program_Error before modifying the vector.

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

A Cursor value is *ambiguous* if any of the following have occurred since it was created:

- Insert, Insert_Space, or Delete has been called on the vector that contains the element the cursor designates with an index value (or a cursor designating an element at such an index value) less than or equal to the index value of the element designated by the cursor; or
- The vector that contains the element it designates has been passed to the Sort or Merge procedures of an instance of Generic_Sorting, or to the Reverse_Elements procedure.

It is a bounded error to call any subprogram other than "=" or Has_Element declared in Containers.Vectors with an ambiguous (but not invalid, see below) cursor parameter. Possible results are:

- The cursor may be treated as if it were No_Element;
- The cursor may designate some element in the vector (but not necessarily the element that it originally designated);
- Constraint_Error may be raised; or
- Program_Error may be raised.

Erroneous Execution

A Cursor value is *invalid* if any of the following have occurred since it was created:

- The vector that contains the element it designates has been finalized;
- The vector that contains the element it designates has been used as the Source or Target of a call to Move; or
- The element it designates has been deleted.

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Vectors is called with an invalid cursor parameter.

Implementation Requirements

No storage associated with a vector object shall be lost upon assignment or scope exit.

The execution of an assignment_statement for a vector shall have the effect of copying the elements from the source vector object to the target vector object.

Implementation Advice

Containers.Vectors should be implemented similarly to an array. In particular, if the length of a vector is *N*, then

- the worst-case time complexity of Element should be $O(\log N)$;
- the worst-case time complexity of Append with Count=1 when *N* is less than the capacity of the vector should be $O(\log N)$; and
- the worst-case time complexity of Prepend with Count=1 and Delete_First with Count=1 should be $O(N \log N)$.

The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be $O(N^{**}2)$, and the average time complexity should be better than $O(N^{**}2)$.

Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements.

Move should not copy elements, and should minimize copying of internal data structures.

If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation.

NOTES

41 All elements of a vector occupy locations in the internal array. If a sparse container is required, a Hashed_Map should be used rather than a vector.

42 If Index_Type'Base'First = Index_Type'First an instance of Ada.Containers.Vectors will raise Constraint_Error. A value below Index_Type'First is required so that an empty vector has a meaningful value of Last_Index.

A.18.3 The Package Containers.Doubly_Linked_Lists

Insert new clause:

The language-defined generic package Containers.Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. A list container is optimized for insertion and deletion at any position.

A doubly-linked list container object manages a linked list of internal *nodes*, each of which contains an element and pointers to the next (successor) and previous (predecessor) internal nodes. A cursor designates a particular node within a list (and by extension the element contained in that node). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved in the container.

The *length* of a list is the number of elements it contains.

Static Semantics

The generic library package Containers.Doubly_Linked_Lists has the following declaration:

```
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   pragma Preelaborate(Doubly_Linked_Lists);

   type List is tagged private;
   pragma Preelaborable_Initialization(List);

   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);

   Empty_List : constant List;

   No_Element : constant Cursor;

   function "=" (Left, Right : List) return Boolean;

   function Length (Container : List) return Count_Type;

   function Is_Empty (Container : List) return Boolean;

   procedure Clear (Container : in out List);

   function Element (Position : Cursor)
      return Element_Type;

   procedure Replace_Element (Container : in out List;
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);

   procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Element : in Element_Type));

   procedure Update_Element
     (Container : in out List;
      Position  : in     Cursor;
      Process   : not null access procedure
                     (Element : in out Element_Type));

   procedure Move (Target : in out List;
                   Source : in out List);

   procedure Insert (Container : in out List;
```

```
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);

procedure Prepend (Container : in out List;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);

procedure Append (Container : in out List;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);

procedure Delete (Container : in out List;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);

procedure Delete_First (Container : in out List;
                        Count     : in     Count_Type := 1);

procedure Delete_Last (Container : in out List;
                       Count     : in     Count_Type := 1);

procedure Reverse_Elements (Container : in out List);

procedure Swap (Container : in out List;
                I, J      : in     Cursor);

procedure Swap_Links (Container : in out List;
                      I, J      : in     Cursor);

procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List);

procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor);

procedure Splice (Container: in out List;
                  Before   : in     Cursor;
                  Position : in     Cursor);

function First (Container : List) return Cursor;

function First_Element (Container : List)
   return Element_Type;

function Last (Container : List) return Cursor;

function Last_Element (Container : List)
   return Element_Type;

function Next (Position : Cursor) return Cursor;

procedure Next (Position : in out Cursor);
```

```
      function Previous (Position : Cursor) return Cursor;

      procedure Previous (Position : in out Cursor);

      function Find (Container : List;
                     Item      : Element_Type;
                     Position  : Cursor := No_Element)
         return Cursor;

      function Reverse_Find (Container : List;
                             Item      : Element_Type;
                             Position  : Cursor := No_Element)
         return Cursor;

      function Contains (Container : List;
                         Item      : Element_Type) return Boolean;

      function Has_Element (Position : Cursor) return Boolean;

      procedure Iterate
        (Container : in List;
         Process   : not null access procedure (Position : in Cursor));

      procedure Reverse_Iterate
        (Container : in List;
         Process   : not null access procedure (Position : in Cursor));

      generic
         with function "<" (Left, Right : Element_Type)
            return Boolean is <>;
      package Generic_Sorting is

         function Is_Sorted (Container : List) return Boolean;

         procedure Sort (Container : in out List);

         procedure Merge (Target  : in out List;
                          Source  : in out List);

      end Generic_Sorting;

   private

      ... -- not specified by the language

   end Ada.Containers.Doubly_Linked_Lists;
```

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions Find, Reverse_Find, and "=" on list values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, and "=" on list values are unspecified.

The type List is used to represent lists. The type List needs finalization (see 7.6).

Empty_List represents the empty List object. It has a length of 0. If an object of type List is not otherwise initialized, it is initialized to the same value as Empty_List.

No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

A subprogram is said to *tamper with cursors* of a list object *L* if:

- it inserts or deletes elements of *L*, that is, it calls the Insert, Clear, Delete, or Delete_Last procedures with *L* as a parameter; or
- it reorders the elements of *L*, that is, it calls the Splice, Swap_Links, or Reverse_Elements procedures or the Sort or Merge procedures of an instance of Generic_Sorting with *L* as a parameter; or
- it finalizes *L*; or
- it calls the Move procedure with *L* as a parameter.

A subprogram is said to *tamper with elements* of a list object *L* if:

- it tampers with cursors of *L*; or
- it replaces one or more elements of *L*, that is, it calls the Replace_Element or Swap procedures with *L* as a parameter.

```
function "=" (Left, Right : List) return Boolean;
```

If Left and Right denote the same list object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise it returns True. Any exception raised during evaluation of element equality is propagated.

```
function Length (Container : List) return Count_Type;
```

Returns the number of elements in Container.

```
function Is_Empty (Container : List) return Boolean;
```

Equivalent to Length (Container) = 0.

```
procedure Clear (Container : in out List);
```

Removes all the elements from Container.

```
function Element (Position : Cursor) return Element_Type;
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

```
procedure Replace_Element (Container : in out List;
                           Position : in    Cursor;
                           New_Item : in    Element_Type);
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element designated by Position.

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

```
procedure Update_Element
  (Container : in out List;
   Position  : in     Cursor;
   Process   : not null access procedure (Element : in out Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

```
procedure Move (Target : in out List;
                Source : in out List);
```

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, the nodes in Source are moved to Target (in the original order). The length of Target is set to the length of Source, and the length of Source is set to 0.

```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert inserts Count copies of New_Item prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert allocates Count copies of New_Item, and inserts them prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last element (if any). Position designates the first newly-inserted element. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert inserts Count new elements prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). The new elements are initialized by default (see 3.3.1). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Prepend (Container : in out List;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, First (Container), New_Item, Count).

```
procedure Append (Container : in out List;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
```

Equivalent to Insert (Container, No_Element, New_Item, Count).

```
procedure Delete (Container : in out List;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);
```

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise Delete removes (from Container) Count elements

starting at the element designated by Position (or all of the elements starting at Position if there are fewer than Count elements starting at Position). Finally, Position is set to No_Element.

```
procedure Delete_First (Container : in out List;
                        Count     : in     Count_Type := 1);
```

Equivalent to Delete (Container, First (Container), Count).

```
procedure Delete_Last (Container : in out List;
                       Count     : in     Count_Type := 1);
```

If Length (Container) <= Count then Delete_Last is equivalent to Clear (Container). Otherwise it removes the last Count nodes from Container.

```
procedure Reverse_Elements (Container : in out List);
```

Reorders the elements of Container in reverse order.

```
procedure Swap (Container : in out List;
                I, J      : in     Cursor);
```

If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

```
procedure Swap_Links (Container : in out List;
                      I, J      : in     Cursor);
```

If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap_Links exchanges the nodes designated by I and J.

```
procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List);
```

If Before is not No_Element, and does not designate an element in Target, then Program_Error is propagated. Otherwise, if Source denotes the same object as Target, the operation has no effect. Otherwise, Splice reorders elements such that they are removed from Source and moved to Target, immediately prior to Before. If Before equals No_Element, the nodes of Source are spliced after the last node of Target. The length of Target is incremented by the number of nodes in Source, and the length of Source is set to 0.

```
procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor);
```

If Position is No_Element then Constraint_Error is propagated. If Before does not equal No_Element, and does not designate an element in Target, then Program_Error is propagated. If Position does not equal No_Element, and does not designate a node in Source, then Program_Error is propagated. If Source denotes the same object as Target, then there is no effect if Position equals Before, else the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. In both cases, Position and the length of Target are unchanged. Otherwise the element designated by Position is removed from Source and moved to Target, immediately prior to Before, or, if Before equals No_Element, after the last element of Target. The length of Target is incremented, the length of Source is decremented, and Position is updated to represent an element in Target.

```
procedure Splice (Container: in out List;
                  Before   : in     Cursor;
                  Position : in     Cursor);
```

If Position is No_Element then Constraint_Error is propagated. If Before does not equal No_Element, and does not designate an element in Container, then Program_Error is propagated. If Position does not equal No_Element, and does not designate a node in Container, then Program_Error is propagated. If Position equals Before there is no effect. Otherwise, the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. The length of Container is unchanged.

```
function First (Container : List) return Cursor;
```

If Container is empty, First returns the value No_Element. Otherwise it returns a cursor that designates the first node in Container.

```
function First_Element (Container : List) return Element_Type;
```

Equivalent to Element (First (Container)).

```
function Last (Container : List) return Cursor;
```

If Container is empty, Last returns the value No_Element. Otherwise it returns a cursor that designates the last node in Container.

```
function Last_Element (Container : List) return Element_Type;
```

Equivalent to Element (Last (Container)).

```
function Next (Position : Cursor) return Cursor;
```

If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the successor of the element designated by Position.

```
procedure Next (Position : in out Cursor);
```

Equivalent to Position := Next (Position).

```
function Previous (Position : Cursor) return Cursor;
```

If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor of the element designated by Position.

```
procedure Previous (Position : in out Cursor);
```

Equivalent to Position := Previous (Position).

```
function Find (Container : List;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
  return Cursor;
```

If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the first element if Position equals No_Element. It proceeds towards Last (Container). If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Reverse_Find (Container : List;
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;
```

If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the last element if Position equals No_Element. It proceeds towards First (Container). If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Contains (Container : List;
                   Item      : Element_Type) return Boolean;
```

Equivalent to Find (Container, Item) /= No_Element.

```
function Has_Element (Position : Cursor) return Boolean;
```

Returns True if Position designates an element, and returns False otherwise.

```
procedure Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));
```

Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor as per the Next function. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

```
procedure Reverse_Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));
```

Iterates over the nodes in Container as per Iterate, except that elements are traversed in reverse order, starting with the last node and moving the cursor as per the Previous function.

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

```
function Is_Sorted (Container : List) return Boolean;
```

Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

```
procedure Sort (Container : in out List);
```

Reorders the nodes of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. The sort is stable. Any exception raised during evaluation of "<" is propagated.

```
procedure Merge (Target  : in out List;
                 Source  : in out List);
```

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

Bounded (Run-Time) Errors

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

Erroneous Execution

A Cursor value is *invalid* if any of the following have occurred since it was created:

- The list that contains the element it designates has been finalized;
- The list that contains the element it designates has been used as the Source or Target of a call to Move; or
- The element it designates has been deleted.

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Doubly_Linked_Lists is called with an invalid cursor parameter.

Implementation Requirements

No storage associated with a doubly-linked List object shall be lost upon assignment or scope exit.

The execution of an assignment_statement for a list shall have the effect of copying the elements from the source list object to the target list object.

Implementation Advice

Containers.Doubly_Linked_Lists should be implemented similarly to a linked list. In particular, if *N* is the length of a list, then the worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 should be *O*(log *N*).

The worst-case time complexity of a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2).

Move should not copy elements, and should minimize copying of internal data structures.

If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation.

NOTES

43 Sorting a list never copies elements, and is a stable sort (equal elements remain in the original order). This is different than sorting an array or vector, which may need to copy elements, and is probably not a stable sort.

A.18.4 Maps

Insert new clause:

The language-defined generic packages Containers.Hashed_Maps and Containers.Ordered_Maps provide private types Map and Cursor, and a set of operations for each type. A map container allows an arbitrary type to be used as a key to find the element associated with that key. A hashed map uses a hash function to organize the keys, while an ordered map orders the keys per a specified relation.

This section describes the declarations that are common to both kinds of maps. See A.18.5 for a description of the semantics specific to Containers.Hashed_Maps and A.18.6 for a description of the semantics specific to Containers.Ordered_Maps.

Static Semantics

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on map values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on map values are unspecified.

The type Map is used to represent maps. The type Map needs finalization (see 7.6).

A map contains pairs of keys and elements, called *nodes*. Map cursors designate nodes, but also can be thought of as designating an element (the element contained in the node) for consistency with the other containers. There exists an equivalence relation on keys, whose definition is different for hashed maps and ordered maps. A map never contains two or more nodes with equivalent keys. The *length* of a map is the number of nodes it contains.

Each nonempty map has two particular nodes called the *first node* and the *last node* (which may be the same). Each node except for the last node has a *successor node*. If there are no other intervening operations, starting with the first node and repeatedly going to the successor node will visit each node in the map exactly once until the last node is reached. The exact definition of these terms is different for hashed maps and ordered maps.

Some operations of these generic packages have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

A subprogram is said to *tamper with cursors* of a map object *M* if:

- it inserts or deletes elements of *M*, that is, it calls the Insert, Include, Clear, Delete, or Exclude procedures with *M* as a parameter; or
- it finalizes *M*; or
- it calls the Move procedure with *M* as a parameter; or
- it calls one of the operations defined to tamper with the cursors of *M*.

A subprogram is said to *tamper with elements* of a map object *M* if:

- it tampers with cursors of *M*; or
- it replaces one or more elements of *M*, that is, it calls the Replace or Replace_Element procedures with *M* as a parameter.

Empty_Map represents the empty Map object. It has a length of 0. If an object of type Map is not otherwise initialized, it is initialized to the same value as Empty_Map.

No_Element represents a cursor that designates no node. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

```
function "=" (Left, Right : Map) return Boolean;
```

If Left and Right denote the same map object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each key *K* in Left, the function returns False if:

- a key equivalent to *K* is not present in Right; or
- the element associated with *K* in Left is not equal to the element associated with *K* in Right (using the generic formal equality operator for elements).

If the function has not returned a result after checking all of the keys, it returns True. Any exception raised during evaluation of key equivalence or element equality is propagated.

```
function Length (Container : Map) return Count_Type;
```

Returns the number of nodes in Container.

```
function Is_Empty (Container : Map) return Boolean;
```

Equivalent to Length (Container) = 0.

```
procedure Clear (Container : in out Map);
```

Removes all the nodes from Container.

```
function Key (Position : Cursor) return Key_Type;
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Key returns the key component of the node designated by Position.

```
function Element (Position : Cursor) return Element_Type;
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element component of the node designated by Position.

```
procedure Replace_Element (Container : in out Map;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns New_Item to the element of the node designated by Position.

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Key     : in Key_Type;
                                         Element : in Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

```
procedure Update_Element
  (Container : in out Map;
   Position  : in     Cursor;
   Process   : not null access procedure (Key     : in     Key_Type;
                                          Element : in out Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

```
procedure Move (Target : in out Map;
                Source : in out Map);
```

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, each node from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

Insert checks if a node with a key equivalent to Key is already present in Container. If a match is found, Inserted is set to False and Position designates the element with the matching key. Otherwise, Insert allocates a new node, initializes it to Key and New_Item, and adds it to Container; Inserted is set to True and Position designates the newly-inserted node. Any exception raised during allocation is propagated and Container is not modified.

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

Insert inserts Key into Container as per the five-parameter Insert, with the difference that an element initialized by default (see 3.3.1) is inserted.

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type);
```

Insert inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then Constraint_Error is propagated.

```
procedure Include (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

Include inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then this operation assigns Key and New_Item to the matching node. Any exception raised during assignment is propagated.

```
procedure Replace (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);
```

Replace checks if a node with a key equivalent to Key is present in Container. If a match is found, Replace assigns Key and New_Item to the matching node; otherwise, Constraint_Error is propagated.

```
procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type);
```

Exclude checks if a node with a key equivalent to Key is present in Container. If a match is found, Exclude removes the node from the map.

```
procedure Delete (Container : in out Map;
                  Key       : in     Key_Type);
```

Delete checks if a node with a key equivalent to Key is present in Container. If a match is found, Delete removes the node from the map; otherwise, Constraint_Error is propagated.

```
procedure Delete (Container : in out Map;
```

```
                    Position  : in out Cursor);
```

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete removes the node designated by Position from the map. Position is set to No_Element on return.

```
function First (Container : Map) return Cursor;
```

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first node in Container.

```
function Next (Position  : Cursor) return Cursor;
```

Returns a cursor that designates the successor of the node designated by Position. If Position designates the last node, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

```
procedure Next (Position  : in out Cursor);
```

Equivalent to Position := Next (Position).

```
function Find (Container : Map;
               Key       : Key_Type) return Cursor;
```

If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if a node with a key equivalent to Key is present in Container. If a match is found, a cursor designating the matching node is returned; otherwise, No_Element is returned.

```
function Element (Container : Map;
                  Key       : Key_Type) return Element_Type;
```

Equivalent to Element (Find (Container, Key)).

```
function Contains (Container : Map;
                   Key       : Key_Type) return Boolean;
```

Equivalent to Find (Container, Key) /= No_Element.

```
function Has_Element (Position : Cursor) return Boolean;
```

Returns True if Position designates a node, and returns False otherwise.

```
procedure Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```

Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor according to the successor relation. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

Erroneous Execution

A Cursor value is *invalid* if any of the following have occurred since it was created:

- The map that contains the node it designates has been finalized;
- The map that contains the node it designates has been used as the Source or Target of a call to Move; or
- The node it designates has been deleted from the map.

The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashed_Maps or Containers.Ordered_Maps is called with an invalid cursor parameter.

Implementation Requirements

No storage associated with a Map object shall be lost upon assignment or scope exit.

The execution of an assignment_statement for a map shall have the effect of copying the elements from the source map object to the target map object.

Implementation Advice

Move should not copy elements, and should minimize copying of internal data structures.

If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

A.18.5 The Package Containers.Hashed_Maps

Insert new clause:

Static Semantics

The generic library package Containers.Hashed_Maps has the following declaration:

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type)
      return Boolean;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Hashed_Maps is
   pragma Preelaborate(Hashed_Maps);

   type Map is tagged private;
   pragma Preelaborable_Initialization(Map);

   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);

   Empty_Map : constant Map;

   No_Element : constant Cursor;

   function "=" (Left, Right : Map) return Boolean;

   function Capacity (Container : Map) return Count_Type;

   procedure Reserve_Capacity (Container : in out Map;
                               Capacity  : in     Count_Type);

   function Length (Container : Map) return Count_Type;

   function Is_Empty (Container : Map) return Boolean;

   procedure Clear (Container : in out Map);

   function Key (Position : Cursor) return Key_Type;

   function Element (Position : Cursor) return Element_Type;

   procedure Replace_Element (Container : in out Map;
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);

   procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Key     : in Key_Type;
                                            Element : in Element_Type));

   procedure Update_Element
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                     (Key     : in     Key_Type;
                      Element : in out Element_Type));

   procedure Move (Target : in out Map;
```

```
                Source : in out Map);

procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);

procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);

procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type);

procedure Include (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);

procedure Replace (Container : in out Map;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type);

procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type);

procedure Delete (Container : in out Map;
                  Key       : in     Key_Type);

procedure Delete (Container : in out Map;
                  Position  : in out Cursor);

function First (Container : Map)
   return Cursor;

function Next (Position  : Cursor) return Cursor;

procedure Next (Position  : in out Cursor);

function Find (Container : Map;
               Key       : Key_Type)
   return Cursor;

function Element (Container : Map;
                  Key       : Key_Type)
   return Element_Type;

function Contains (Container : Map;
                   Key       : Key_Type) return Boolean;

function Has_Element (Position : Cursor) return Boolean;

function Equivalent_Keys (Left, Right : Cursor)
   return Boolean;

function Equivalent_Keys (Left  : Cursor;
                          Right : Key_Type)
   return Boolean;

function Equivalent_Keys (Left  : Key_Type;
                          Right : Cursor)
   return Boolean;

procedure Iterate
  (Container : in Map;
```

```
      Process   : not null access procedure (Position : in Cursor));

private

   ... -- not specified by the language

end Ada.Containers.Hashed_Maps;
```

An object of type Map contains an expandable hash table, which is used to provide direct access to nodes. The *capacity* of an object of type Map is the maximum number of nodes that can be inserted into the hash table prior to it being automatically expanded.

Two keys *K1* and *K2* are defined to be *equivalent* if Equivalent_Keys (*K1*, *K2*) returns True.

The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular key value. For any two equivalent key values, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

The actual function for the generic formal function Equivalent_Keys on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Keys behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Keys, and how many times they call it, is unspecified.

If the value of a key stored in a node of a map is changed other than by an operation in this package such that at least one of Hash or Equivalent_Keys give different results, the behavior of this package is unspecified.

Which nodes are the first node and the last node of a map, and which node is the successor of a given node, are unspecified, other than the general semantics described in A.18.4.

```
function Capacity (Container : Map) return Count_Type;
```

Returns the capacity of Container.

```
procedure Reserve_Capacity (Container : in out Map;
                            Capacity  : in     Count_Type);
```

Reserve_Capacity allocates a new hash table such that the length of the resulting map can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the nodes in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified.

Reserve_Capacity tampers with the cursors of Container.

```
procedure Clear (Container : in out Map);
```

In addition to the semantics described in A.18.4, Clear does not affect the capacity of Container.

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

In addition to the semantics described in A.18.4, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value.

```
function Equivalent_Keys (Left, Right : Cursor)
      return Boolean;
```

Equivalent to Equivalent_Keys (Key (Left), Key (Right)).

```
function Equivalent_Keys (Left  : Cursor;
                          Right : Key_Type) return Boolean;
```

Equivalent to Equivalent_Keys (Key (Left), Right).


```
function Equivalent_Keys (Left  : Key_Type;
                          Right : Cursor) return Boolean;
```

Equivalent to Equivalent_Keys (Left, Key (Right)).

Implementation Advice

If *N* is the length of a map, the average time complexity of the subprograms Element, Insert, Include, Replace, Delete, Exclude and Find that take a key parameter should be *O*(log *N*). The average time complexity of the subprograms that take a cursor parameter should be *O*(1). The average time complexity of Reserve_Capacity should be *O*(*N*).

A.18.6 The Package Containers.Ordered_Maps

Insert new clause:

Static Semantics

The generic library package Containers.Ordered_Maps has the following declaration:

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right : Key_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps is
   pragma Preelaborate(Ordered_Maps);

   function Equivalent_Keys (Left, Right : Key_Type) return Boolean;

   type Map is tagged private;
   pragma Preelaborable_Initialization(Map);

   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);

   Empty_Map : constant Map;

   No_Element : constant Cursor;

   function "=" (Left, Right : Map) return Boolean;

   function Length (Container : Map) return Count_Type;

   function Is_Empty (Container : Map) return Boolean;

   procedure Clear (Container : in out Map);

   function Key (Position : Cursor) return Key_Type;

   function Element (Position : Cursor) return Element_Type;

   procedure Replace_Element (Container : in out Map;
                              Position  : in     Cursor;
                              New_Item  : in     Element_Type);

   procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Key     : in Key_Type;
                                            Element : in Element_Type));

   procedure Update_Element
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                     (Key     : in     Key_Type;
                      Element : in out Element_Type));
```

```
   procedure Move (Target : in out Map;
                   Source : in out Map);

   procedure Insert (Container : in out Map;
                     Key       : in     Key_Type;
                     New_Item  : in     Element_Type;
                     Position  :    out Cursor;
                     Inserted  :    out Boolean);

   procedure Insert (Container : in out Map;
                     Key       : in     Key_Type;
                     Position  :    out Cursor;
                     Inserted  :    out Boolean);

   procedure Insert (Container : in out Map;
                     Key       : in     Key_Type;
                     New_Item  : in     Element_Type);

   procedure Include (Container : in out Map;
                      Key       : in     Key_Type;
                      New_Item  : in     Element_Type);

   procedure Replace (Container : in out Map;
                      Key       : in     Key_Type;
                      New_Item  : in     Element_Type);

   procedure Exclude (Container : in out Map;
                      Key       : in     Key_Type);

   procedure Delete (Container : in out Map;
                     Key       : in     Key_Type);

   procedure Delete (Container : in out Map;
                     Position  : in out Cursor);

   procedure Delete_First (Container : in out Map);

   procedure Delete_Last (Container : in out Map);

   function First (Container : Map) return Cursor;

   function First_Element (Container : Map) return Element_Type;

   function First_Key (Container : Map) return Key_Type;

   function Last (Container : Map) return Cursor;

   function Last_Element (Container : Map) return Element_Type;

   function Last_Key (Container : Map) return Key_Type;

   function Next (Position : Cursor) return Cursor;

   procedure Next (Position : in out Cursor);

   function Previous (Position : Cursor) return Cursor;

   procedure Previous (Position : in out Cursor);

   function Find (Container : Map;
                  Key       : Key_Type) return Cursor;

   function Element (Container : Map;
                     Key       : Key_Type) return Element_Type;

   function Floor (Container : Map;
                   Key       : Key_Type) return Cursor;
```

```
   function Ceiling (Container : Map;
                     Key       : Key_Type) return Cursor;

   function Contains (Container : Map;
                      Key       : Key_Type) return Boolean;

   function Has_Element (Position : Cursor) return Boolean;

   function "<" (Left, Right : Cursor) return Boolean;

   function ">" (Left, Right : Cursor) return Boolean;

   function "<" (Left : Cursor; Right : Key_Type) return Boolean;

   function ">" (Left : Cursor; Right : Key_Type) return Boolean;

   function "<" (Left : Key_Type; Right : Cursor) return Boolean;

   function ">" (Left : Key_Type; Right : Cursor) return Boolean;

   procedure Iterate
     (Container : in Map;
      Process   : not null access procedure (Position : in Cursor));

   procedure Reverse_Iterate
     (Container : in Map;
      Process   : not null access procedure (Position : in Cursor));

private

   ... -- not specified by the language

end Ada.Containers.Ordered_Maps;
```

Two keys *K1* and *K2* are *equivalent* if both *K1* < *K2* and *K2* < *K1* return False, using the generic formal "<" operator for keys. Function Equivalent_Keys returns True if Left and Right are equivalent, and False otherwise.

The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

If the value of a key stored in a map is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

The first node of a nonempty map is the one whose key is less than the key of all the other nodes in the map. The last node of a nonempty map is the one whose key is greater than the key of all the other elements in the map. The successor of a node is the node with the smallest key that is larger than the key of the given node. The predecessor of a node is the node with the largest key that is smaller than the key of the given node. All comparisons are done using the generic formal "<" operator for keys.

```
procedure Delete_First (Container : in out Map);
```

If Container is empty, Delete_First has no effect. Otherwise the node designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

```
procedure Delete_Last (Container : in out Map);
```

If Container is empty, Delete_Last has no effect. Otherwise the node designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

```
function First_Element (Container : Map) return Element_Type;
```

Equivalent to Element (First (Container)).

```
function First_Key (Container : Map) return Key_Type;
```

Equivalent to Key (First (Container)).

```
function Last (Container : Map) return Cursor;
```

Returns a cursor that designates the last node in Container. If Container is empty, returns No_Element.

```
function Last_Element (Container : Map) return Element_Type;
```

Equivalent to Element (Last (Container)).

```
function Last_Key (Container : Map) return Key_Type;
```

Equivalent to Key (Last (Container)).

```
function Previous (Position : Cursor) return Cursor;
```

If Position equals No_Element, then Previous returns No_Element. Otherwise Previous returns a cursor designating the node that precedes the one designated by Position. If Position designates the first element, then Previous returns No_Element.

```
procedure Previous (Position : in out Cursor);
```

Equivalent to Position := Previous (Position).

```
function Floor (Container : Map;
                Key       : Key_Type) return Cursor;
```

Floor searches for the last node whose key is not greater than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise No_Element is returned.

```
function Ceiling (Container : Map;
                  Key       : Key_Type) return Cursor;
```

Ceiling searches for the first node whose key is not less than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise No_Element is returned.

```
function "<" (Left, Right : Cursor) return Boolean;
```

Equivalent to Key (Left) < Key (Right).

```
function ">" (Left, Right : Cursor) return Boolean;
```

Equivalent to Key (Right) < Key (Left).

```
function "<" (Left : Cursor; Right : Key_Type) return Boolean;
```

Equivalent to Key (Left) < Right.

```
function ">" (Left : Cursor; Right : Key_Type) return Boolean;
```

Equivalent to Right < Key (Left).

```
function "<" (Left : Key_Type; Right : Cursor) return Boolean;
```

Equivalent to Left < Key (Right).

```
function ">" (Left : Key_Type; Right : Cursor) return Boolean;
```

Equivalent to Key (Right) < Left.

```
procedure Reverse_Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```

Iterates over the nodes in Container as per Iterate, with the difference that the nodes are traversed in predecessor order, starting with the last node.

Implementation Advice

If *N* is the length of a map, then the worst-case time complexity of the Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1).

A.18.7 Sets

Insert new clause:

The language-defined generic packages Containers.Hashed_Sets and Containers.Ordered_Sets provide private types Set and Cursor, and a set of operations for each type. A set container allows elements of an arbitrary type to be stored without duplication. A hashed set uses a hash function to organize elements, while an ordered set orders its element per a specified relation.

This section describes the declarations that are common to both kinds of sets. See A.18.8 for a description of the semantics specific to Containers.Hashed_Sets and A.18.9 for a description of the semantics specific to Containers.Ordered_Sets.

Static Semantics

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on set values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on set values are unspecified.

The type Set is used to represent sets. The type Set needs finalization (see 7.6).

A set contains elements. Set cursors designate elements. There exists an equivalence relation on elements, whose definition is different for hashed sets and ordered sets. A set never contains two or more equivalent elements. The *length* of a set is the number of elements it contains.

Each nonempty set has two particular elements called the *first element* and the *last element* (which may be the same). Each element except for the last element has a *successor element*. If there are no other intervening operations, starting with the first element and repeatedly going to the successor element will visit each element in the set exactly once until the last element is reached. The exact definition of these terms is different for hashed sets and ordered sets.

Some operations of these generic packages have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

A subprogram is said to *tamper with cursors* of a set object *S* if:

- it inserts or deletes elements of *S*, that is, it calls the Insert, Include, Clear, Delete, Exclude, or Replace_Element procedures with *S* as a parameter; or
- it finalizes *S*; or
- it calls the Move procedure with *S* as a parameter; or
- it calls one of the operations defined to tamper with cursors of *S*.

A subprogram is said to *tamper with elements* of a set object *S* if:

- it tampers with cursors of *S*.

Empty_Set represents the empty Set object. It has a length of 0. If an object of type Set is not otherwise initialized, it is initialized to the same value as Empty_Set.

No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

```
function "=" (Left, Right : Set) return Boolean;
```

If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equal to *E* (using the generic formal equality operator) is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equality is propagated.

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```

If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equivalent to *E* is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equivalence is propagated.

```
function To_Set (New_Item : Element_Type) return Set;
```

Returns a set containing the single element New_Item.

```
function Length (Container : Set) return Count_Type;
```

Returns the number of elements in Container.

```
function Is_Empty (Container : Set) return Boolean;
```

Equivalent to Length (Container) = 0.

```
procedure Clear (Container : in out Set);
```

Removes all the elements from Container.

```
function Element (Position : Cursor) return Element_Type;
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

```
procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. If an element equivalent to New_Item is already present in Container at a position other than Position, Program_Error is propagated. Otherwise, Replace_Element assigns New_Item to the element designated by Position. Any exception raised by the assignment is propagated.

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

```
procedure Move (Target : in out Set;
                Source : in out Set);
```

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first clears Target. Then, each element from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

Insert checks if an element equivalent to New_Item is already present in Container. If a match is found, Inserted is set to False and Position designates the matching element. Otherwise, Insert adds New_Item to Container; Inserted is set to True and Position designates the newly-inserted element. Any exception raised during allocation is propagated and Container is not modified.

```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type);
```

Insert inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then Constraint_Error is propagated.

```
procedure Include (Container : in out Set;
                   New_Item  : in     Element_Type);
```

Include inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then it is replaced. Any exception raised during assignment is propagated.

```
procedure Replace (Container : in out Set;
                   New_Item  : in     Element_Type);
```

Replace checks if an element equivalent to New_Item is already in the set. If a match is found, that element is replaced with New_Item; otherwise, Constraint_Error is propagated.

```
procedure Exclude (Container : in out Set;
                   Item      : in     Element_Type);
```

Exclude checks if an element equivalent to Item is present in Container. If a match is found, Exclude removes the element from the set.

```
procedure Delete (Container : in out Set;
                  Item      : in     Element_Type);
```

Delete checks if an element equivalent to Item is present in Container. If a match is found, Delete removes the element from the set; otherwise, Constraint_Error is propagated.

```
procedure Delete (Container : in out Set;
                  Position  : in out Cursor);
```

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete removes the element designated by Position from the set. Position is set to No_Element on return.

```
procedure Union (Target : in out Set;
                 Source : in     Set);
```

Union inserts into Target the elements of Source that are not equivalent to some element already in Target.

```
function Union (Left, Right : Set) return Set;
```

Returns a set comprising all of the elements of Left, and the elements of Right that are not equivalent to some element of Left.

```
procedure Intersection (Target : in out Set;
                        Source : in     Set);
```

Union deletes from Target the elements of Target that are not equivalent to some element of Source.

```
function Intersection (Left, Right : Set) return Set;
```

Returns a set comprising all the elements of Left that are equivalent to the some element of Right.

```
procedure Difference (Target : in out Set;
                      Source : in     Set);
```

If Target denotes the same object as Source, then Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source.

```
function Difference (Left, Right : Set) return Set;
```

Returns a set comprising the elements of Left that are not equivalent to some element of Right.

```
procedure Symmetric_Difference (Target : in out Set;
                                Source : in     Set);
```

If Target denotes the same object as Source, then Symmetric_Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source, and inserts into Target the elements of Source that are not equivalent to some element of Target.

```
function Symmetric_Difference (Left, Right : Set) return Set;
```

Returns a set comprising the elements of Left that are not equivalent to some element of Right, and the elements of Right that are not equivalent to some element of Left.

```
function Overlap (Left, Right : Set) return Boolean;
```

If an element of Left is equivalent to some element of Right, then Overlap returns True. Otherwise it returns False.

```
function Is_Subset (Subset : Set;
                    Of_Set : Set) return Boolean;
```

If an element of Subset is not equivalent to some element of Of_Set, then Is_Subset returns False. Otherwise it returns True.

```
function First (Container : Set) return Cursor;
```

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first element in Container.

```
function Next (Position  : Cursor) return Cursor;
```

Returns a cursor that designates the successor of the element designated by Position. If Position designates the last element, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

```
procedure Next (Position  : in out Cursor);
```

Equivalent to Position := Next (Position).

```
function Find (Container : Set;
               Item      : Element_Type) return Cursor;
```

If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if an element equivalent to Item is present in Container. If a match is found, a cursor designating the matching element is returned; otherwise, No_Element is returned.

```
function Contains (Container : Set;
                   Item      : Element_Type) return Boolean;
```

Equivalent to Find (Container, Item) /= No_Element.

```
function Has_Element (Position : Cursor) return Boolean;
```

Returns True if Position designates an element, and returns False otherwise.

```
procedure Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));
```

Iterate calls Process.**all** with a cursor that designates each element in Container, starting with the first element and moving the cursor according to the successor relation. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

Both Containers.Hashed_Set and Containers.Ordered_Set declare a nested generic package Generic_Keys, which provides operations that allow set manipulation in terms of a key (typically, a portion of an element) instead of a complete element. The formal function Key of Generic_Keys extracts a key value from an element. It is expected to return the same value each time it is called with a particular element. The behavior of Generic_Keys is unspecified if Key behaves in some other manner.

A key is expected to unambiguously determine a single equivalence class for elements. The behavior of Generic_Keys is unspecified if the formal parameters of this package behave in some other manner.

```
function Key (Position : Cursor) return Key_Type;
```

Equivalent to Key (Element (Position)).

The subprograms in package Generic_Keys named Contains, Find, Element, Delete, and Exclude, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key parameter is used to locate an element in the set.

```
procedure Replace (Container : in out Set;
                   Key       : in     Key_Type;
```

```
                    New_Item  : in     Element_Type);
```

Equivalent to Replace_Element (Container, Find (Container, Key), New_Item).

```
procedure Update_Element_Preserving_Key
  (Container : in out Set;
   Position  : in     Cursor;
   Process   : not null access procedure
                               (Element : in out Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Update_Element_Preserving_Key uses Key to save the key value *K* of the element designated by Position. Update_Element_Preserving_Key then calls Process.**all** with that element as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated. After Process.**all** returns, Update_Element_Preserving_Key checks if *K* determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

Erroneous Execution

A Cursor value is *invalid* if any of the following have occurred since it was created:

- The set that contains the element it designates has been finalized;
- The set that contains the element it designates has been used as the Source or Target of a call to Move; or
- The element it designates has been deleted from the set.

The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashed_Sets or Containers.Ordered_Sets is called with an invalid cursor parameter.

Implementation Requirements

No storage associated with a Set object shall be lost upon assignment or scope exit.

The execution of an assignment_statement for a set shall have the effect of copying the elements from the source set object to the target set object.

Implementation Advice

Move should not copy elements, and should minimize copying of internal data structures.

If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation.

A.18.8 The Package Containers.Hashed_Sets

Insert new clause:

Static Semantics

The generic library package Containers.Hashed_Sets has the following declaration:

```
generic
   type Element_Type is private;
   with function Hash (Element : Element_Type) return Hash_Type;
   with function Equivalent_Elements (Left, Right : Element_Type)
                 return Boolean;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Sets is
   pragma Preelaborate(Hashed_Sets);

   type Set is tagged private;
   pragma Preelaborable_Initialization(Set);
```

```
type Cursor is private;
pragma Preelaborable_Initialization(Cursor);

Empty_Set : constant Set;

No_Element : constant Cursor;

function "=" (Left, Right : Set) return Boolean;

function Equivalent_Sets (Left, Right : Set) return Boolean;

function To_Set (New_Item : Element_Type) return Set;

function Capacity (Container : Set) return Count_Type;

procedure Reserve_Capacity (Container : in out Set;
                            Capacity  : in     Count_Type);

function Length (Container : Set) return Count_Type;

function Is_Empty (Container : Set) return Boolean;

procedure Clear (Container : in out Set);

function Element (Position : Cursor) return Element_Type;

procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);

procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));

procedure Move (Target : in out Set;
                Source : in out Set);

procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);

procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type);

procedure Include (Container : in out Set;
                   New_Item  : in     Element_Type);

procedure Replace (Container : in out Set;
                   New_Item  : in     Element_Type);

procedure Exclude (Container : in out Set;
                   Item      : in     Element_Type);

procedure Delete (Container : in out Set;
                  Item      : in     Element_Type);

procedure Delete (Container : in out Set;
                  Position  : in out Cursor);

procedure Union (Target : in out Set;
                 Source : in     Set);

function Union (Left, Right : Set) return Set;

function "or" (Left, Right : Set) return Set renames Union;
```

```
procedure Intersection (Target : in out Set;
                        Source : in     Set);

function Intersection (Left, Right : Set) return Set;

function "and" (Left, Right : Set) return Set renames Intersection;

procedure Difference (Target : in out Set;
                      Source : in     Set);

function Difference (Left, Right : Set) return Set;

function "-" (Left, Right : Set) return Set renames Difference;

procedure Symmetric_Difference (Target : in out Set;
                                Source : in     Set);

function Symmetric_Difference (Left, Right : Set) return Set;

function "xor" (Left, Right : Set) return Set
  renames Symmetric_Difference;

function Overlap (Left, Right : Set) return Boolean;

function Is_Subset (Subset : Set;
                    Of_Set : Set) return Boolean;

function First (Container : Set) return Cursor;

function Next (Position : Cursor) return Cursor;

procedure Next (Position : in out Cursor);

function Find (Container : Set;
               Item      : Element_Type) return Cursor;

function Contains (Container : Set;
                   Item      : Element_Type) return Boolean;

function Has_Element (Position : Cursor) return Boolean;

function Equivalent_Elements (Left, Right : Cursor)
  return Boolean;

function Equivalent_Elements (Left  : Cursor;
                              Right : Element_Type)
  return Boolean;

function Equivalent_Elements (Left  : Element_Type;
                              Right : Cursor)
  return Boolean;

procedure Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));

generic
   type Key_Type (<>) is private;
   with function Key (Element : Element_Type) return Key_Type;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type) return Boolean;
package Generic_Keys is

   function Key (Position : Cursor) return Key_Type;

   function Element (Container : Set;
                     Key       : Key_Type)
```

```
      return Element_Type;

   procedure Replace (Container : in out Set;
                      Key       : in     Key_Type;
                      New_Item  : in     Element_Type);

   procedure Exclude (Container : in out Set;
                      Key       : in     Key_Type);

   procedure Delete (Container : in out Set;
                     Key       : in     Key_Type);

   function Find (Container : Set;
                  Key       : Key_Type)
      return Cursor;

   function Contains (Container : Set;
                      Key       : Key_Type)
      return Boolean;

   procedure Update_Element_Preserving_Key
     (Container : in out Set;
      Position  : in     Cursor;
      Process   : not null access procedure
                      (Element : in out Element_Type));

end Generic_Keys;

private

   ... -- not specified by the language

end Ada.Containers.Hashed_Sets;
```

An object of type Set contains an expandable hash table, which is used to provide direct access to elements. The *capacity* of an object of type Set is the maximum number of elements that can be inserted into the hash table prior to it being automatically expanded.

Two elements *E1* and *E2* are defined to be *equivalent* if Equivalent_Elements (*E1*, *E2*) returns True.

The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular element value. For any two equivalent elements, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

The actual function for the generic formal function Equivalent_Elements is expected to return the same value each time it is called with a particular pair of Element values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Elements behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Elements, and how many times they call it, is unspecified.

If the value of an element stored in a set is changed other than by an operation in this package such that at least one of Hash or Equivalent_Elements give different results, the behavior of this package is unspecified.

Which elements are the first element and the last element of a set, and which element is the successor of a given element, are unspecified, other than the general semantics described in A.18.7.

```
function Capacity (Container : Set) return Count_Type;
```

Returns the capacity of Container.

```
procedure Reserve_Capacity (Container : in out Set;
                            Capacity  : in     Count_Type);
```

Reserve_Capacity allocates a new hash table such that the length of the resulting set can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the elements in Container onto the new hash table. It

replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified.

Reserve_Capacity tampers with the cursors of Container.

```
procedure Clear (Container : in out Set);
```

In addition to the semantics described in A.18.7, Clear does not affect the capacity of Container.

```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

In addition to the semantics described in A.18.7, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value.

```
function First (Container : Set) return Cursor;
```

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first hashed element in Container.

```
function Equivalent_Elements (Left, Right : Cursor)
      return Boolean;
```

Equivalent to Equivalent_Elements (Element (Left), Element (Right)).

```
function Equivalent_Elements (Left  : Cursor;
                              Right : Element_Type) return Boolean;
```

Equivalent to Equivalent_Elements (Element (Left), Right).

```
function Equivalent_Elements (Left  : Element_Type;
                              Right : Cursor) return Boolean;
```

Equivalent to Equivalent_Elements (Left, Element (Right)).

For any element *E*, the actual function for the generic formal function Generic_Keys.Hash is expected to be such that Hash (*E*) = Generic_Keys.Hash (Key (*E*)). If the actuals for Key or Generic_Keys.Hash behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Generic_Keys.Hash, and how many times they call it, is unspecified.

For any two elements *E1* and *E2*, the boolean values Equivalent_Elements (*E1*, *E2*) and Equivalent_Keys (Key (*E1*), Key (*E2*)) are expected to be equal. If the actuals for Key or Equivalent_Keys behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Equivalent_Keys, and how many times they call it, is unspecified.

Implementation Advice

If *N* is the length of a set, the average time complexity of the subprograms Insert, Include, Replace, Delete, Exclude and Find that take an element parameter should be $O(\log N)$. The average time complexity of the subprograms that take a cursor parameter should be $O(1)$. The average time complexity of Reserve_Capacity should be $O(N)$.

A.18.9 The Package Containers.Ordered_Sets

Insert new clause:

Static Semantics

The generic library package Containers.Ordered_Sets has the following declaration:

```
generic
   type Element_Type is private;
   with function "<" (Left, Right : Element_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
   pragma Preelaborate(Ordered_Sets);

   function Equivalent_Elements (Left, Right : Element_Type) return Boolean;
```

```
type Set is tagged private;
pragma Preelaborable_Initialization(Set);

type Cursor is private;
pragma Preelaborable_Initialization(Cursor);

Empty_Set : constant Set;

No_Element : constant Cursor;

function "=" (Left, Right : Set) return Boolean;

function Equivalent_Sets (Left, Right : Set) return Boolean;

function To_Set (New_Item : Element_Type) return Set;

function Length (Container : Set) return Count_Type;

function Is_Empty (Container : Set) return Boolean;

procedure Clear (Container : in out Set);

function Element (Position : Cursor) return Element_Type;

procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);

procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));

procedure Move (Target : in out Set;
                Source : in out Set);

procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);

procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type);

procedure Include (Container : in out Set;
                   New_Item  : in     Element_Type);

procedure Replace (Container : in out Set;
                   New_Item  : in     Element_Type);

procedure Exclude (Container : in out Set;
                   Item      : in     Element_Type);

procedure Delete (Container : in out Set;
                  Item      : in     Element_Type);

procedure Delete (Container : in out Set;
                  Position  : in out Cursor);

procedure Delete_First (Container : in out Set);

procedure Delete_Last (Container : in out Set);

procedure Union (Target : in out Set;
                 Source : in     Set);

function Union (Left, Right : Set) return Set;
```

```
function "or" (Left, Right : Set) return Set renames Union;

procedure Intersection (Target : in out Set;
                        Source : in     Set);

function Intersection (Left, Right : Set) return Set;

function "and" (Left, Right : Set) return Set renames Intersection;

procedure Difference (Target : in out Set;
                      Source : in     Set);

function Difference (Left, Right : Set) return Set;

function "-" (Left, Right : Set) return Set renames Difference;

procedure Symmetric_Difference (Target : in out Set;
                                Source : in     Set);

function Symmetric_Difference (Left, Right : Set) return Set;

function "xor" (Left, Right : Set) return Set renames
   Symmetric_Difference;

function Overlap (Left, Right : Set) return Boolean;

function Is_Subset (Subset : Set;
                    Of_Set : Set) return Boolean;

function First (Container : Set) return Cursor;

function First_Element (Container : Set) return Element_Type;

function Last (Container : Set) return Cursor;

function Last_Element (Container : Set) return Element_Type;

function Next (Position : Cursor) return Cursor;

procedure Next (Position : in out Cursor);

function Previous (Position : Cursor) return Cursor;

procedure Previous (Position : in out Cursor);

function Find (Container : Set;
               Item      : Element_Type)
   return Cursor;

function Floor (Container : Set;
                Item      : Element_Type)
   return Cursor;

function Ceiling (Container : Set;
                  Item      : Element_Type)
   return Cursor;

function Contains (Container : Set;
                   Item      : Element_Type) return Boolean;

function Has_Element (Position : Cursor) return Boolean;

function "<" (Left, Right : Cursor) return Boolean;

function ">" (Left, Right : Cursor) return Boolean;
```

```
function "<" (Left : Cursor; Right : Element_Type)
   return Boolean;

function ">" (Left : Cursor; Right : Element_Type)
   return Boolean;

function "<" (Left : Element_Type; Right : Cursor)
   return Boolean;

function ">" (Left : Element_Type; Right : Cursor)
   return Boolean;

procedure Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));

procedure Reverse_Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));

generic
   type Key_Type (<>) is private;
   with function Key (Element : Element_Type) return Key_Type;
   with function "<" (Left, Right : Key_Type)
      return Boolean is <>;
package Generic_Keys is

    function Equivalent_Keys (Left, Right : Key_Type)
       return Boolean;

    function Key (Position : Cursor) return Key_Type;

    function Element (Container : Set;
                      Key       : Key_Type)
       return Element_Type;

    procedure Replace (Container : in out Set;
                       Key       : in     Key_Type;
                       New_Item  : in     Element_Type);

    procedure Exclude (Container : in out Set;
                       Key       : in     Key_Type);

    procedure Delete (Container : in out Set;
                      Key       : in     Key_Type);

    function Find (Container : Set;
                   Key       : Key_Type)
       return Cursor;

    function Floor (Container : Set;
                    Key       : Key_Type)
       return Cursor;

    function Ceiling (Container : Set;
                      Key       : Key_Type)
       return Cursor;

    function Contains (Container : Set;
                       Key       : Key_Type) return Boolean;

    procedure Update_Element_Preserving_Key
      (Container : in out Set;
       Position  : in     Cursor;
       Process   : not null access procedure
                          (Element : in out Element_Type));
```

```
   end Generic_Keys;

private

   ... -- not specified by the language

end Ada.Containers.Ordered_Sets;
```

Two elements *E1* and *E2* are *equivalent* if both *E1* < *E2* and *E2* < *E1* return False, using the generic formal "<" operator for elements. Function Equivalent_Elements returns True if Left and Right are equivalent, and False otherwise.

The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

If the value of an element stored in a set is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

The first element of a nonempty set is the one which is less than all the other elements in the set. The last element of a nonempty set is the one which is greater than all the other elements in the set. The successor of an element is the smallest element that is larger than the given element. The predecessor of an element is the largest element that is smaller than the given element. All comparisons are done using the generic formal "<" operator for elements.

```
procedure Delete_First (Container : in out Set);
```

If Container is empty, Delete_First has no effect. Otherwise the element designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

```
procedure Delete_Last (Container : in out Set);
```

If Container is empty, Delete_Last has no effect. Otherwise the element designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

```
function First_Element (Container : Set) return Element_Type;
```

Equivalent to Element (First (Container)).

```
function Last (Container : Set) return Cursor;
```

Returns a cursor that designates the last element in Container. If Container is empty, returns No_Element.

```
function Last_Element (Container : Set) return Element_Type;
```

Equivalent to Element (Last (Container)).

```
function Previous (Position : Cursor) return Cursor;
```

If Position equals No_Element, then Previous returns No_Element. Otherwise Previous returns a cursor designating the element that precedes the one designated by Position. If Position designates the first element, then Previous returns No_Element.

```
procedure Previous (Position : in out Cursor);
```

Equivalent to Position := Previous (Position).

```
function Floor (Container : Set;
                Item      : Element_Type) return Cursor;
```

Floor searches for the last element which is not greater than Item. If such an element is found, a cursor that designates it is returned. Otherwise No_Element is returned.

```
function Ceiling (Container : Set;
                  Item      : Element_Type) return Cursor;
```

Ceiling searches for the first element which is not less than Item. If such an element is found, a cursor that designates it is returned. Otherwise No_Element is returned.

```
function "<" (Left, Right : Cursor) return Boolean;
```

Equivalent to Element (Left) < Element (Right).

```
function ">" (Left, Right : Cursor) return Boolean;
```

Equivalent to Element (Right) < Element (Left).

```
function "<" (Left : Cursor; Right : Element_Type) return Boolean;
```

Equivalent to Element (Left) < Right.

```
function ">" (Left : Cursor; Right : Element_Type) return Boolean;
```

Equivalent to Right < Element (Left).

```
function "<" (Left : Element_Type; Right : Cursor) return Boolean;
```

Equivalent to Left < Element (Right).

```
function ">" (Left : Element_Type; Right : Cursor) return Boolean;
```

Equivalent to Element (Right) < Left.

```
procedure Reverse_Iterate
   (Container : in Set;
    Process   : not null access procedure (Position : in Cursor));
```

Iterates over the elements in Container as per Iterate, with the difference that the elements are traversed in predecessor order, starting with the last element.

For any two elements *E1* and *E2*, the boolean values (*E1* < *E2*) and (Key(*E1*) < Key(*E2*)) are expected to be equal. If the actuals for Key or Generic_Keys."<" behave in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Key and Generic_Keys."<", and how many times the functions are called, is unspecified.

In addition to the semantics described in A.18.7, the subprograms in package Generic_Keys named Floor and Ceiling, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key subprogram parameter is compared to elements in the container using the Key and "<" generic formal functions. The function named Equivalent_Keys in package Generic_Keys returns True if both Left < Right and Right < Left return False using the generic formal "<" operator, and returns True otherwise.

Implementation Advice

If *N* is the length of a set, then the worst-case time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations that take an element parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1).

A.18.10 The Package Containers.Indefinite_Vectors

Insert new clause:

The language-defined generic package Containers.Indefinite_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Vectors has the same contents as Containers.Vectors except:

- The generic formal Element_Type is indefinite.
- The procedures with the profiles:

```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

are omitted.

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

A.18.11 The Package Containers.Indefinite_Doubly_Linked_Lists

Insert new clause:

The language-defined generic package Containers.Indefinite_Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Doubly_Linked_Lists has the same contents as Containers.Doubly_Linked_Lists except:

- The generic formal Element_Type is indefinite.
- The procedure with the profile:

```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

  is omitted.

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

A.18.12 The Package Containers.Indefinite_Hashed_Maps

Insert new clause:

The language-defined generic package Containers.Indefinite_Hashed_Maps provides a map with the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the generic formal types Key_Type and Element_Type are indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Hashed_Maps has the same contents as Containers.Hashed_Maps except:

- The generic formal Key_Type is indefinite.
- The generic formal Element_Type is indefinite.
- The procedure with the profile:

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

  is omitted.

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

A.18.13 The Package Containers.Indefinite_Ordered_Maps

Insert new clause:

The language-defined generic package Containers.Indefinite_Ordered_Maps provides a map with the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the generic formal types Key_Type and Element_Type are indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Ordered_Maps has the same contents as Containers.Ordered_Maps except:

- The generic formal Key_Type is indefinite.
- The generic formal Element_Type is indefinite.
- The procedure with the profile:

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

  is omitted.

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

A.18.14 The Package Containers.Indefinite_Hashed_Sets

Insert new clause:

The language-defined generic package Containers.Indefinite_Hashed_Sets provides a set with the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Hashed_Sets has the same contents as Containers.Hashed_Sets except:

- The generic formal Element_Type is indefinite.
- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

A.18.15 The Package Containers.Indefinite_Ordered_Sets

Insert new clause:

The language-defined generic package Containers.Indefinite_Ordered_Sets provides a set with the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Ordered_Sets has the same contents as Containers.Ordered_Sets except:

- The generic formal Element_Type is indefinite.
- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

A.18.16 Array Sorting

Insert new clause:

The language-defined generic procedures Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort provide sorting on arbitrary array types.

Static Semantics

The generic library procedure Containers.Generic_Array_Sort has the following declaration:

```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type range <>) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort (Container : in out Array_Type);
pragma Pure(Ada.Containers.Generic_Array_Sort);
```

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

The actual function for the generic formal function "<" of Generic_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Array_Sort is unspecified. How many times Generic_Array_Sort calls "<" is unspecified.

The generic library procedure Containers.Generic_Constrained_Array_Sort has the following declaration:

```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Constrained_Array_Sort
      (Container : in out Array_Type);
pragma Pure(Ada.Containers.Generic_Constrained_Array_Sort);
```

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

The actual function for the generic formal function "<" of Generic_Constrained_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Constrained_Array_Sort is unspecified. How many times Generic_Constrained_Array_Sort calls "<" is unspecified.

Implementation Advice

The worst-case time complexity of a call on an instance of Containers.Generic_Array_Sort or Containers.Generic_Constrained_Array_Sort should be $O(N**2)$ or better, and the average time complexity should be better than $O(N**2)$, where N is the length of the Container parameter.

Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements.

Annex B: Interface to Other Languages

B.1 Interfacing Pragmas

Insert after paragraph 38:

Notwithstanding what this International Standard says elsewhere, the elaboration of a declaration denoted by the local_name of a pragma Import does not create the entity. Such an elaboration has no other effect than to allow the defining name to denote the external entity.

the new paragraph:

Erroneous Execution

It is the programmer's responsibility to ensure that the use of interfacing pragmas does not violate Ada semantics; otherwise, program execution is erroneous.

Delete paragraph 49:

8 An interfacing pragma might result in an effect that violates Ada semantics.

B.2 The Package Interfaces

Insert after paragraph 10:

- Floating point types corresponding to each floating point format fully supported by the hardware.

the new paragraph:

Support for interfacing to any foreign language is optional. However, an implementation shall not provide any attribute, library unit, or pragma having the same name as an attribute, library unit, or pragma (respectively) specified in the following clauses of this Annex unless the provided construct is either as specified in those clauses or is more limited in capability than that required by those clauses. A program that attempts to use an unsupported capability of this Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

Insert after paragraph 11:

An implementation may provide implementation-defined library units that are children of Interfaces, and may add declarations to the visible part of Interfaces in addition to the ones defined above.

the new paragraph:

A child package of package Interfaces with the name of a convention may be provided independently of whether the convention is supported by the pragma Convention and vice versa. Such a child package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces.

Delete paragraph 12:

For each implementation-defined convention identifier, there should be a child package of package Interfaces with the corresponding name. This package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces.

B.3 Interfacing with C and C++

Replace the title:

Interfacing with C

by:

Interfacing with C and C++

Replace paragraph 1:

The facilities relevant to interfacing with the C language are the package Interfaces.C and its children; support for the Import, Export, and Convention pragmas with *convention*_identifier C; and support for the Convention pragma with *convention*_identifier C_Pass_By_Copy.

by:

The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children; support for the Import, Export, and Convention pragmas with *convention*_identifier C; and support for the Convention pragma with *convention*_identifier C_Pass_By_Copy.

Replace paragraph 2:

The package Interfaces.C contains the basic types, constants and subprograms that allow an Ada program to pass scalars and strings to C functions.

by:

The package Interfaces.C contains the basic types, constants and subprograms that allow an Ada program to pass scalars and strings to C and C++ functions. When this clause mentions a C entity, the reference also applies to the corresponding entity in C++.

Insert after paragraph 39:

```
procedure To_Ada (Item     : in wchar_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in Boolean := True);
```

the new paragraphs:

```
-- ISO/IEC 10646:2003 compatible types defined by ISO/IEC TR 19769:2004.

type char16_t is <implementation-defined character type>;

char16_nul : constant char16_t := implementation-defined;

function To_C (Item : in Wide_Character) return char16_t;
function To_Ada (Item : in char16_t) return Wide_Character;

type char16_array is array (size_t range <>) of aliased char16_t;

pragma Pack(char16_array);

function Is_Nul_Terminated (Item : in char16_array) return Boolean;
function To_C (Item       : in Wide_String;
               Append_Nul : in Boolean := True)
   return char16_array;

function To_Ada (Item     : in char16_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out char16_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char16_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);

type char32_t is <implementation-defined character type>;

char32_nul : constant char32_t := implementation-defined;
```

```
function To_C (Item : in Wide_Wide_Character) return char32_t;
function To_Ada (Item : in char32_t) return Wide_Wide_Character;

type char32_array is array (size_t range <>) of aliased char32_t;

pragma Pack(char32_array);

function Is_Nul_Terminated (Item : in char32_array) return Boolean;
function To_C (Item       : in Wide_Wide_String;
               Append_Nul : in Boolean := True)
   return char32_array;

function To_Ada (Item     : in char32_array;
                 Trim_Nul : in Boolean := True)
   return Wide_Wide_String;

procedure To_C (Item       : in  Wide_Wide_String;
                Target     : out char32_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char32_array;
                  Target   : out Wide_Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

Replace paragraph 43:

The types int, short, long, unsigned, ptrdiff_t, size_t, double, char, and wchar_t correspond respectively to the C types having the same names. The types signed_char, unsigned_short, unsigned_long, unsigned_char, C_float, and long_double correspond respectively to the C types signed char, unsigned short, unsigned long, unsigned char, float, and long double.

by:

The types int, short, long, unsigned, ptrdiff_t, size_t, double, char, wchar_t, char16_t, and char32_t correspond respectively to the C types having the same names. The types signed_char, unsigned_short, unsigned_long, unsigned_char, C_float, and long_double correspond respectively to the C types signed char, unsigned short, unsigned long, unsigned char, float, and long double.

Replace paragraph 50:

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True.

by:

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.

Insert after paragraph 60:

The To_C and To_Ada subprograms that convert between Wide_String and wchar_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that wide_nul is used instead of nul.

the new paragraphs:

```
function Is_Nul_Terminated (Item : in char16_array) return Boolean;
```

The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

```
function To_C   (Item : in Wide_Character) return char16_t;
function To_Ada (Item : in char16_t      ) return Wide_Character;
```

To_C and To_Ada provide mappings between the Ada and C 16-bit character types.

```
function To_C (Item       : in Wide_String;
               Append_Nul : in Boolean := True)
   return char16_array;

function To_Ada (Item     : in char16_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out char16_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char16_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

The To_C and To_Ada subprograms that convert between Wide_String and char16_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char16_nul is used instead of nul.

```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
```

The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

```
function To_C   (Item : in Wide_Wide_Character) return char32_t;
function To_Ada (Item : in char32_t           ) return Wide_Wide_Character;
```

To_C and To_Ada provide mappings between the Ada and C 32-bit character types.

```
function To_C (Item       : in Wide_Wide_String;
               Append_Nul : in Boolean := True)
   return char32_array;

function To_Ada (Item     : in char32_array;
                 Trim_Nul : in Boolean := True)
   return Wide_Wide_String;

procedure To_C (Item       : in  Wide_Wide_String;
                Target     : out char32_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char32_array;
                  Target   : out Wide_Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

The To_C and To_Ada subprograms that convert between Wide_Wide_String and char32_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char32_nul is used instead of nul.

Replace paragraph 60.2:

The eligibility rules in B.1 do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible.

by:

The eligibility rules in B.1 do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is an unchecked union type or if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible.

Replace paragraph 62.1:

The constants nul and wide_nul should have a representation of zero.

by:

The constants nul, wide_nul, char16_nul, and char32_nul should have a representation of zero.

Replace paragraph 68.1:

- An Ada parameter of a C_Pass_By_Copy-compatible (record) type T, of mode **in**, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

by:

- An Ada parameter of a (record) type T of convention C_Pass_By_Copy, of mode **in**, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

Replace paragraph 69:

- An Ada parameter of a record type T, of any mode, other than an **in** parameter of a C_Pass_By_Copy-compatible type, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

by:

- An Ada parameter of a record type T, of any mode, other than an **in** parameter of a type of convention C_Pass_By_Copy, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

Insert after paragraph 71:

- An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.

the new paragraph:

An Ada parameter of a private type is passed as specified for the full view of the type.

Delete paragraph 74:

11 There is no explicit support for C's union types. Unchecked conversions can be used to obtain the effect of C unions.

B.3.1 The Package Interfaces.C.Strings

Replace paragraph 5:

```
type chars_ptr is private;
```

by:

```
type chars_ptr is private;
pragma Preelaborable_Initialization(chars_ptr);
```

Replace paragraph 6:

```
type chars_ptr_array is array (size_t range <>) of chars_ptr;
```

by:

```
type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;
```

Replace paragraph 50:

Equivalent to Update(Item, Offset, To_C(Str), Check).

by:

Equivalent to Update(Item, Offset, To_C(Str, Append_Nul => False), Check).

B.3.3 Pragma Unchecked_Union

Insert new clause:

A pragma Unchecked_Union specifies an interface correspondence between a given discriminated type and some C union. The pragma specifies that the associated type shall be given a representation that leaves no space for its discriminant(s).

Syntax

The form of a pragma Unchecked_Union is as follows:

pragma Unchecked_Union (*first_subtype*_local_name);

Legality Rules

Unchecked_Union is a representation pragma, specifying the unchecked union aspect of representation.

The *first_subtype*_local_name of a pragma Unchecked_Union shall denote an unconstrained discriminated record subtype having a variant_part.

A type to which a pragma Unchecked_Union applies is called an *unchecked union type*. A subtype of an unchecked union type is defined to be an *unchecked union subtype*. An object of an unchecked union type is defined to be an *unchecked union object*.

All component subtypes of an unchecked union type shall be C-compatible.

If a component subtype of an unchecked union type is subject to a per-object constraint, then the component subtype shall be an unchecked union subtype.

Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type.

A component declared in a variant_part of an unchecked union type shall not have a controlled, protected, or task part.

The completion of an incomplete or private type declaration having a known_discriminant_part shall not be an unchecked union type.

An unchecked union subtype shall only be passed as a generic actual parameter if the corresponding formal type has no known discriminants or is an unchecked union type.

Static Semantics

An unchecked union type is eligible for convention C.

All objects of an unchecked union type have the same size.

Discriminants of objects of an unchecked union type are of size zero.

Any check which would require reading a discriminant of an unchecked union object is suppressed (see 11.5). These checks include:

- The check performed when addressing a variant component (i.e., a component that was declared in a variant part) of an unchecked union object that the object has this component (see 4.1.3).
- Any checks associated with a type or subtype conversion of a value of an unchecked union type (see 4.6). This includes, for example, the check associated with the implicit subtype conversion of an assignment statement.
- The subtype membership check associated with the evaluation of a qualified expression (see 4.7) or an uninitialized allocator (see 4.8).

Dynamic Semantics

A view of an unchecked union object (including a type conversion or function call) has *inferable discriminants* if it has a constrained nominal subtype, unless the object is a component of an enclosing unchecked union object that is subject to a per-object constraint and the enclosing object lacks inferable discriminants.

An expression of an unchecked union type has inferable discriminants if it is either a name of an object with inferable discriminants or a qualified expression whose subtype_mark denotes a constrained subtype.

Program_Error is raised in the following cases:

- Evaluation of the predefined equality operator for an unchecked union type if either of the operands lacks inferable discriminants.
- Evaluation of the predefined equality operator for a type which has a subcomponent of an unchecked union type whose nominal subtype is unconstrained.
- Evaluation of a membership test if the subtype_mark denotes a constrained unchecked union subtype and the expression lacks inferable discriminants.
- Conversion from a derived unchecked union type to an unconstrained non-unchecked-union type if the operand of the conversion lacks inferable discriminants.
- Execution of the default implementation of the Write or Read attribute of an unchecked union type.
- Execution of the default implementation of the Output or Input attribute of an unchecked union type if the type lacks default discriminant values.

Implementation Permissions

An implementation may require that pragma Controlled be specified for the type of an access subcomponent of an unchecked union type.

NOTES

15 The use of an unchecked union to obtain the effect of an unchecked conversion results in erroneous execution (see 11.5). Execution of the following example is erroneous even if Float'Size = Integer'Size:

```
type T (Flag : Boolean := False) is
   record
       case Flag is
           when False =>
               F1 : Float := 0.0;
           when True =>
               F2 : Integer := 0;
       end case;
    end record;
pragma Unchecked_Union (T);

X : T;
Y : Integer := X.F2; -- erroneous
```

Annex C: Systems Programming

C.2 Required Representation Support

Replace paragraph 1:

This clause specifies minimal requirements on the implementation's support for representation items and related features.

by:

This clause specifies minimal requirements on the support for representation items and related features.

C.3 Interrupt Support

Replace paragraph 23:

If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required as part of the execution of subprograms of a protected object whose one of its subprograms is an interrupt handler.

by:

If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required as part of the execution of subprograms of a protected object for which one of its subprograms is an interrupt handler.

Replace paragraph 26:

Other forms of handlers are allowed to be supported, in which case, the rules of this subclause should be adhered to.

by:

Other forms of handlers are allowed to be supported, in which case the rules of this clause should be adhered to.

Replace paragraph 28:

If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for a finer-grain control of interrupt blocking.

by:

If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for finer-grained control of interrupt blocking.

C.3.1 Protected Procedure Handlers

Replace paragraph 7:

The Attach_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library level declaration.

by:

The Attach_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration.

Replace paragraph 8:

The Interrupt_Handler pragma is only allowed immediately within a protected_definition. The corresponding protected_type_declaration shall be a library level declaration. In addition, any object_declaration of such a type shall be a library level declaration.

by:

The Interrupt_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration.

Replace paragraph 11:

If the Ceiling_Locking policy (see D.3) is in effect then upon the initialization of a protected object that either an Attach_Handler or Interrupt_Handler pragma applies to one of its procedures, a check is made that the ceiling priority defined in the protected_definition is in the range of System.Interrupt_Priority. If the check fails, Program_Error is raised.

by:

If the Ceiling_Locking policy (see D.3) is in effect, then upon the initialization of a protected object for which either an Attach_Handler or Interrupt_Handler pragma applies to one of its procedures, a check is made that the ceiling priority defined in the protected_definition is in the range of System.Interrupt_Priority. If the check fails, Program_Error is raised.

Replace paragraph 16:

1. The worst case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C – (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.

by:

- The worst-case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C – (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.

Replace paragraph 23:

5 The ceiling priority of a protected object that one of its procedures is attached to an interrupt should be at least as high as the highest processor priority at which that interrupt will ever be delivered.

by:

5 A protected object that has a (protected) procedure attached to an interrupt should have a ceiling priority at least as high as the highest processor priority at which that interrupt will ever be delivered.

C.3.2 The Package Interrupts

Replace paragraph 22:

The Reference function returns a value of type System.Address that can be used to attach a task entry, via an address clause (see J.7.1) to the interrupt specified by Interrupt. This function raises Program_Error if attaching task entries to interrupts (or to this particular interrupt) is not supported.

by:

The Reference function returns a value of type System.Address that can be used to attach a task entry via an address clause (see J.7.1) to the interrupt specified by Interrupt. This function raises Program_Error if attaching task entries to interrupts (or to this particular interrupt) is not supported.

Replace paragraph 24:

If the Ceiling_Locking policy (see D.3) is in effect the implementation shall document the default ceiling priority assigned to a protected object that contains either the Attach_Handler or Interrupt_Handler pragmas, but not the Interrupt_Priority pragma. This default need not be the same for all interrupts.

by:

If the Ceiling_Locking policy (see D.3) is in effect, the implementation shall document the default ceiling priority assigned to a protected object that contains either the Attach_Handler or Interrupt_Handler pragmas, but not the Interrupt_Priority pragma. This default need not be the same for all interrupts.

C.4 Preelaboration Requirements

Insert after paragraph 4:

- Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any;

the new paragraph:

- no subtype_mark denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type;

C.5 Pragma Discard_Names

Replace paragraph 7:

If the pragma applies to an enumeration type, then the semantics of the Wide_Image and Wide_Value attributes are implementation defined for that type; the semantics of Image and Value are still defined in terms of Wide_Image and Wide_Value. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the pragma applies to a tagged type, then the semantics of the Tags.Expanded_Name function are implementation defined for that type. If the pragma applies to an exception, then the semantics of the Exceptions.Exception_Name function are implementation defined for that exception.

by:

If the pragma applies to an enumeration type, then the semantics of the Wide_Wide_Image and Wide_Wide_Value attributes are implementation defined for that type; the semantics of Image, Wide_Image, Value, and Wide_Value are still defined in terms of Wide_Wide_Image and Wide_Wide_Value. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the pragma applies to a tagged type, then the semantics of the Tags.Wide_Wide_Expanded_Name function are implementation defined for that type; the semantics of Tags.Expanded_Name and Tags.Wide_Expanded_Name are still defined in terms of Tags.Wide_Wide_Expanded_Name. If the pragma applies to an exception, then the semantics of the Exceptions.Wide_Wide_Exception_Name function are implementation defined for that exception; the semantics of Exceptions.Exception_Name and Exceptions.Wide_Exception_Name are still defined in terms of Exceptions.Wide_Wide_Exception_Name.

C.6 Shared Variable Control

Replace paragraph 7:

An *atomic* type is one to which a pragma Atomic applies. An *atomic* object (including a component) is one to which a pragma Atomic applies, or a component of an array to which a pragma Atomic_Components applies, or any object of an atomic type.

by:

An *atomic* type is one to which a pragma Atomic applies. An *atomic* object (including a component) is one to which a pragma Atomic applies, or a component of an array to which a pragma Atomic_Components applies, or any object of an atomic type, other than objects obtained by evaluating a slice.

Insert after paragraph 21:

If a pragma Pack applies to a type any of whose subcomponents are atomic, the implementation shall not pack the atomic subcomponents more tightly than that for which it can support indivisible reads and updates.

the new paragraphs:

Implementation Advice

A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others.

A load or store of an atomic object should, where possible, be implemented by a single load or store instruction.

C.7 Task Identification and Attributes

Replace the title:

Task Identification and Attributes

by:

Task Information

Replace paragraph 1:

This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined.

by:

This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined.

C.7.1 The Package Task_Identification

Replace paragraph 2:

```
package Ada.Task_Identification is
   type Task_ID is private;
   Null_Task_ID : constant Task_ID;
   function  "=" (Left, Right : Task_ID) return Boolean;
```

by:

```
package Ada.Task_Identification is
   pragma Preelaborate(Task_Identification);
   type Task_Id is private;
   pragma Preelaborable_Initialization (Task_Id);
   Null_Task_Id : constant Task_Id;
   function  "=" (Left, Right : Task_Id) return Boolean;
```

Replace paragraph 17:

It is a bounded error to call the Current_Task function from an entry body or an interrupt handler. Program_Error is raised, or an implementation-defined value of the type Task_ID is returned.

by:

It is a bounded error to call the Current_Task function from an entry body, interrupt handler, or finalization of a task attribute. Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

C.7.2 The Package Task_Attributes

Insert after paragraph 13:

For all the operations declared in this package, Tasking_Error is raised if the task identified by T is terminated. Program_Error is raised if the value of T is Null_Task_ID.

the new paragraph:

After a task has terminated, all of its attributes are finalized, unless they have been finalized earlier. When the master of an instantiation of Ada.Task_Attributes is finalized, the corresponding attribute of each task is finalized, unless it has been finalized earlier.

Replace paragraph 15.1:

Accesses to task attributes via a value of type Attribute_Handle are erroneous if executed concurrently with each other or with calls of any of the operations declared in package Task_Attributes.

by:

An access to a task attribute via a value of type Attribute_Handle is erroneous if executed concurrently with another such access or a call of any of the operations declared in package Task_Attributes. An access to a task attribute is erroneous if executed concurrently with or after the finalization of the task attribute.

Replace paragraph 17:

When a task terminates, an implementation shall finalize all attributes of the task, and reclaim any other storage associated with the attributes.

by:

After task attributes are finalized, the implementation shall reclaim any storage associated with the attributes.

Replace paragraph 20:

The implementation shall document the following metrics: A task calling the following subprograms shall execute in a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task T are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the Attribute type shall be a scalar whose size is equal to the size of the predefined integer size. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task (that is, the default value for the T parameter is used), and the other, where T identifies another, non-terminated, task.

by:

The implementation shall document the following metrics: A task calling the following subprograms shall execute at a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task T are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the Attribute type shall be a scalar type whose size is equal to the size of the predefined type Integer. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task (that is, the default value for the T parameter is used), and the other, where T identifies another, non-terminated, task.

Replace paragraph 26:

- a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value.

by:

- a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value;

Replace paragraph 30:

Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from

the heap. This can be accomplished by either placing restrictions on the number and the size of the task's attributes, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.

by:

Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the attributes of a task, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.

Delete paragraph 33:

14 As specified in C.7.1, if the parameter T (in a call on a subprogram of an instance of this package) identifies a nonexistent task, the execution of the program is erroneous.

C.7.3 The Package Task_Termination

Insert new clause:

Static Semantics

The following language-defined library package exists:

```
with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination is
   pragma Preelaborate(Task_Termination);

   type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);

   type Termination_Handler is access protected procedure
     (Cause : in Cause_Of_Termination;
      T     : in Ada.Task_Identification.Task_Id;
      X     : in Ada.Exceptions.Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler
     (Handler: in Termination_Handler);
   function Current_Task_Fallback_Handler return Termination_Handler;

   procedure Set_Specific_Handler
     (T       : in Ada.Task_Identification.Task_Id;
      Handler : in Termination_Handler);
   function Specific_Handler (T : Ada.Task_Identification.Task_Id)
      return Termination_Handler;

end Ada.Task_Termination;
```

Dynamic Semantics

The type Termination_Handler identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a *handler*. In all cases T identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, then Cause is set to Normal and X is set to Null_Occurrence. If the task terminates because it is being aborted, then Cause is set to Abnormal and X is set to Null_Occurrence. If the task terminates because of an exception raised by the execution of its task_body, then Cause is set to Unhandled_Exception and X is set to the associated exception occurrence.

Each task has two termination handlers, a *fall-back handler* and a *specific handler*. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be *set* if it is associated with a non-null value of type Termination_Handler, and *cleared* otherwise. When a task is created, its specific handler and fall-back handler are cleared.

The procedure Set_Dependents_Fallback_Handler changes the fall-back handler for the calling task; if Handler is **null**, that fall-back handler is cleared, otherwise it is set to be Handler.**all**. If a fall-back handler had previously been set it is replaced.

The function Current_Task_Fallback_Handler returns the fall-back handler that is currently set for the calling task, if one is set; otherwise it returns **null**.

The procedure Set_Specific_Handler changes the specific handler for the task identified by T; if Handler is **null**, that specific handler is cleared, otherwise it is set to be Handler.**all**. If a specific handler had previously been set it is replaced.

The function Specific_Handler returns the specific handler that is currently set for the task identified by T, if one is set; otherwise it returns **null**.

As part of the finalization of a task_body, after performing the actions specified in 7.6 for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed.

For Set_Specific_Handler or Specific_Handler, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Ada.Task_Identification.Null_Task_Id.

An exception propagated from a handler that is invoked as part of the termination of a task has no effect.

Erroneous Execution

For a call of Set_Specific_Handler or Specific_Handler, if the task identified by T no longer exists, the execution of the program is erroneous.

Annex D: Real-Time Systems

D.1 Task Priorities

Replace paragraph 20:

At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see D.11), its base priority is always a source of priority inheritance. Other sources of priority inheritance are specified under the following conditions:

by:

At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see D.11), its base priority is a source of priority inheritance unless otherwise specified for a particular task dispatching policy. Other sources of priority inheritance are specified under the following conditions:

D.2 Priority Scheduling

Replace paragraph 1:

This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9.2). The rules have two parts: the task dispatching model (see D.2.1), and a specific task dispatching policy (see D.2.2).

by:

This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9).

D.2.1 The Task Dispatching Model

Replace paragraph 1:

The task dispatching model specifies preemptive scheduling, based on conceptual priority-ordered ready queues.

by:

The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.

Static Semantics

The following language-defined library package exists:

```
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

Dispatching serves as the parent of other language-defined library units concerned with task dispatching.

Replace paragraph 2:

A task runs (that is, it becomes a *running task*) only when it is ready (see 9.2) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

by:

A task can become a *running task* only if it is ready (see 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

Replace paragraph 4:

Task dispatching is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called *task dispatching points*. A task reaches a task dispatching point whenever it becomes blocked, and whenever it becomes ready. In addition, the completion of an accept_statement (see

9.5.2), and task termination are task dispatching points for the executing task. Other task dispatching points are defined throughout this Annex.

by:

Task dispatching is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called *task dispatching points*. A task reaches a task dispatching point whenever it becomes blocked, and when it terminates. Other task dispatching points are defined throughout this Annex for specific policies.

Replace paragraph 5:

Task dispatching policies are specified in terms of conceptual *ready queues*, task states, and task preemption. A ready queue is an ordered list of ready tasks. The first position in a queue is called the *head of the queue*, and the last position is called the *tail of the queue*. A task is *ready* if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

by:

Task dispatching policies are specified in terms of conceptual *ready queues* and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the *head of the queue*, and the last position is called the *tail of the queue*. A task is *ready* if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

Replace paragraph 6:

Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point, one task is selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

by:

Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

Delete paragraph 7:

A preemptible resource is a resource that while allocated to one task can be allocated (temporarily) to another instead. Processors are preemptible resources. Access to a protected object (see 9.5.1) is a nonpreemptible resource. {preempted task} When a higher-priority task is dispatched to the processor, and the previously running task is placed on the appropriate ready queue, the latter task is said to be *preempted*.

Delete paragraph 8:

A new running task is also selected whenever there is a nonempty ready queue with a higher priority than the priority of the running task, or when the task dispatching policy requires a running task to go back to a ready queue. These are also task dispatching points.

Replace paragraph 9:

An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation defined effect on task dispatching (see D.2.2).

by:

An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching.

Insert after paragraph 10:

An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range.

the new paragraph:

For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation-defined manner. However, a delay_statement always corresponds to at least one task dispatching point.

Insert after paragraph 16:

12 The priority of a task is determined by rules specified in this subclause, and under D.1, "Task Priorities", D.3, "Priority Ceiling Locking", and D.5, "Dynamic Priorities".

the new paragraph:

13 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

D.2.2 Task Dispatching Pragmas

Replace the title:

The Standard Task Dispatching Policy

by:

Task Dispatching Pragmas

Insert before paragraph 1:

Syntax

The form of a pragma Task_Dispatching_Policy is as follows:

the new paragraph:

This clause allows a single task dispatching policy to be defined for all priorities, or the range of priorities to be split into subranges that are assigned individual dispatching policies.

Insert after paragraph 2:

pragma Task_Dispatching_Policy (*policy*_identifier);

the new paragraphs:

The form of a pragma Priority_Specific_Dispatching is as follows:

pragma Priority_Specific_Dispatching (

*policy*_identifier, *first_priority*_expression, *last_priority*_expression);

Name Resolution Rules

The expected type for *first_priority*_expression and *last_priority*_expression is Integer.

Replace paragraph 3:

The *policy*_identifier shall either be FIFO_Within_Priorities or an implementation-defined identifier.

by:

The *policy*_identifier used in a pragma Task_Dispatching_Policy shall be the name of a task dispatching policy.

The *policy*_identifier used in a pragma Priority_Specific_Dispatching shall be the name of a task dispatching policy.

Both *first_priority_*expression and *last_priority_*expression shall be static expressions in the range of System.Any_Priority; *last_priority_*expression shall have a value greater than or equal to *first_priority_*expression.

Static Semantics

Pragma Task_Dispatching_Policy specifies the single task dispatching policy.

Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Tasks with base priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma have their active priorities determined according to the specified dispatching policy. Tasks with active priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma are dispatched according to the specified dispatching policy.

If a partition contains one or more Priority_Specific_Dispatching pragmas the dispatching policy for priorities not covered by any Priority_Specific_Dispatching pragmas is FIFO_Within_Priorities.

Replace paragraph 4:

A Task_Dispatching_Policy pragma is a configuration pragma.

by:

A Task_Dispatching_Policy pragma is a configuration pragma. A Priority_Specific_Dispatching pragma is a configuration pragma.

The priority ranges specified in more than one Priority_Specific_Dispatching pragma within the same partition shall not be overlapping.

If a partition contains one or more Priority_Specific_Dispatching pragmas it shall not contain a Task_Dispatching_Policy pragma.

Delete paragraph 5:

If the FIFO_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

Replace paragraph 6:

A *task dispatching policy* specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues, and whether a task is inserted at the head or the tail of the queue for its active priority. The task dispatching policy is specified by a Task_Dispatching_Policy configuration pragma. If no such pragma appears in any of the program units comprising a partition, the task dispatching policy for that partition is unspecified.

by:

A *task dispatching policy* specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues. A single task dispatching policy is specified by a Task_Dispatching_Policy pragma. Pragma Priority_Specific_Dispatching assigns distinct dispatching policies to subranges of System.Any_Priority.

If neither pragma applies to any of the program units comprising a partition, the task dispatching policy for that partition is unspecified.

If a partition contains one or more Priority_Specific_Dispatching pragmas a task dispatching point occurs for the currently running task of a processor whenever there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

A task that has its base priority changed may move from one dispatching policy to another. It is immediately subject to the new dispatching policy.

Delete paragraph 7:

The language defines only one task dispatching policy, FIFO_Within_Priorities; when this policy is in effect, modifications to the ready queues occur only as follows:

Delete paragraph 8:

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

Delete paragraph 9:

- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.

Delete paragraph 10:

- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

Delete paragraph 11:

- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

Delete paragraph 12:

Each of the events specified above is a task dispatching point (see D.2.1).

Replace paragraph 13:

In addition, when a task is preempted, it is added at the head of the ready queue for its active priority.

by:

Implementation Requirements

An implementation shall allow, for a single partition, both the locking policy (see D.3) to be specified as Ceiling_Locking and also one or more Priority_Specific_Dispatching pragmas to be given.

Delete paragraph 14:

Priority inversion is the duration for which a task remains at the head of the highest priority ready queue while the processor executes a lower priority task. The implementation shall document:

Delete paragraph 15:

- The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

Delete paragraph 16:

- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

Replace paragraph 17:

Implementations are allowed to define other task dispatching policies, but need not support more than one such policy per partition.

by:

Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

An implementation need not support pragma Priority_Specific_Dispatching if it is infeasible to support it in the target environment.

Replace paragraph 18:

For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation defined manner. However, a *delay_statement* always corresponds to at least one task dispatching point.

by:

An implementation need not support pragma Priority_Specific_Dispatching if it is infeasible to support it in the target environment.

Delete paragraph 19:

13 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

Delete paragraph 20:

14 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

Delete paragraph 21:

15 Setting the base priority of a ready task causes the task to move to the end of the queue for its active priority, regardless of whether the active priority of the task actually changes.

D.2.3 Preemptive Dispatching

Insert new clause:

This clause defines a preemptive task dispatching policy.

Static Semantics

The *policy*_identifier FIFO_Within_Priorities is a task dispatching policy.

Dynamic Semantics

When FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.
- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.
- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

Each of the events specified above is a task dispatching point (see D.2.1).

A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be *preempted* and it is added at the head of the ready queue for its active priority.

Implementation Requirements

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

Documentation Requirements

Priority inversion is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

- The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

NOTES

14 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

15 Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.

D.2.4 Non-Preemptive Dispatching

Insert new clause:

This clause defines a non-preemptive task dispatching policy.

Static Semantics

The *policy*_identifier Non_Preemptive_FIFO_Within_Priorities is a task dispatching policy.

Legality Rules

Non_Preemptive_FIFO_Within_Priorities shall not be specified as the *policy*_identifier of pragma Priority_Specific_Dispatching (see D.2.2).

Dynamic Semantics

When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority.
- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

For this policy, a non-blocking delay_statement is the only non-blocking event that is a task dispatching point (see D.2.1).

Implementation Requirements

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Non_Preemptive_FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

Implementation Permissions

Since implementations are allowed to round all ceiling priorities in subrange System.Priority to System.Priority'Last (see D.3), an implementation may allow a task to execute within a protected object without raising its active priority provided the associated protected unit does not contain pragma Interrupt_Priority, Interrupt_Handler, or Attach_Handler.

D.2.5 Round Robin Dispatching

Insert new clause:

This clause defines the task dispatching policy Round_Robin_Within_Priorities and the package Round_Robin.

Static Semantics

The *policy*_identifier Round_Robin_Within_Priorities is a task dispatching policy.

The following language-defined library package exists:

```
with System;
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin is
```



```
   Default_Quantum : constant Ada.Real_Time.Time_Span :=
             implementation-defined;
   procedure Set_Quantum (Pri     : in System.Priority;
                          Quantum : in Ada.Real_Time.Time_Span);
   procedure Set_Quantum (Low, High : in System.Priority;
                          Quantum   : in Ada.Real_Time.Time_Span);
   function Actual_Quantum (Pri : System.Priority) return Ada.Real_Time.Time_Span;
   function Is_Round_Robin (Pri : System.Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

When task dispatching policy Round_Robin_Within_Priorities is the single policy in effect for a partition, each task with priority in the range of System.Interrupt_Priority is dispatched according to policy FIFO_Within_Priorities.

Dynamic Semantics

The procedures Set_Quantum set the required Quantum value for a single priority level Pri or a range of priority levels Low .. High. If no quantum is set for a Round Robin priority level, Default_Quantum is used.

The function Actual_Quantum returns the actual quantum used by the implementation for the priority level Pri.

The function Is_Round_Robin returns True if priority Pri is covered by task dispatching policy Round_Robin_Within_Priorities; otherwise it returns False.

A call of Actual_Quantum or Set_Quantum raises exception Dispatching.Dispatching_Policy_Error if a predefined policy other than Round_Robin_Within_Priorities applies to the specified priority or any of the priorities in the specified range.

For Round_Robin_Within_Priorities, the dispatching rules for FIFO_Within_Priorities apply with the following additional rules:

- When a task is added or moved to the tail of the ready queue for its base priority, it has an execution time budget equal to the quantum for that priority level. This will also occur when a blocked task becomes executable again.
- When a task is preempted (by a higher priority task) and is added to the head of the ready queue for its priority level, it retains its remaining budget.
- While a task is executing, its budget is decreased by the amount of execution time it uses. The accuracy of this accounting is the same as that for execution time clocks (see D.14).
- When a task has exhausted its budget and is without an inherited priority (and is not executing within a protected operation), it is moved to the tail of the ready queue for its priority level. This is a task dispatching point.

Implementation Requirements

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Round_Robin_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

Documentation Requirements

An implementation shall document the quantum values supported.

An implementation shall document the accuracy with which it detects the exhaustion of the budget of a task.

NOTES

17 Due to implementation constraints, the quantum value returned by Actual_Quantum might not be identical to that set with Set_Quantum.

18 A task that executes continuously with an inherited priority will not be subject to round robin dispatching.

D.2.6 Earliest Deadline First Dispatching

Insert new clause:

The deadline of a task is an indication of the urgency of the task; it represents a point on an ideal physical time line. The deadline might affect how resources are allocated to the task.

This clause defines a package for representing the deadline of a task and a dispatching policy that defines Earliest Deadline First (EDF) dispatching. A pragma is defined to assign an initial deadline to a task.

Syntax

The form of a pragma Relative_Deadline is as follows:

pragma Relative_Deadline (*relative_deadline*_expression);

Name Resolution Rules

The expected type for *relative_deadline*_expression is Real_Time.Time_Span.

Legality Rules

A Relative_Deadline pragma is allowed only immediately within a task_definition or the declarative_part of a subprogram_body. At most one such pragma shall appear within a given construct.

Static Semantics

The *policy*_identifier EDF_Across_Priorities is a task dispatching policy.

The following language-defined library package exists:

```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline :=
              Ada.Real_Time.Time_Last;
  procedure Set_Deadline (D : in Deadline;
              T : in Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline (
              Delay_Until_Time : in Ada.Real_Time.Time;
              Deadline_Offset : in Ada.Real_Time.Time_Span);
  function Get_Deadline (T : Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

Post-Compilation Rules

If the EDF_Across_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

If the EDF_Across_Priorities policy appears in a Priority_Specific_Dispatching pragma (see D.2.2) in a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

Dynamic Semantics

A Relative_Deadline pragma has no effect if it occurs in the declarative_part of the subprogram_body of a subprogram other than the main subprogram.

The initial absolute deadline of a task containing pragma Relative_Deadline is the value of Real_Time.Clock + *relative_deadline*_expression, where the call of Real_Time.Clock is made between task creation and the start of its activation. If there is no Relative_Deadline pragma then the initial absolute deadline of a task is the value of Default_Deadline. The environment task is also given an initial deadline by this rule.

The procedure Set_Deadline changes the absolute deadline of the task to D. The function Get_Deadline returns the absolute deadline of the task.

The procedure Delay_Until_And_Set_Deadline delays the calling task until time Delay_Until_Time. When the task becomes runnable again it will have deadline Delay_Until_Time + Deadline_Offset.

On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered on to the ready queue determined by the rules defined below.

When EDF_Across_Priorities is specified for priority range *Low..High* all ready queues in this range are ordered by deadline. The task at the head of a queue is the one with the earliest deadline.

A task dispatching point occurs for the currently running task *T* to which policy EDF_Across_Priorities applies:

- when a change to the deadline of *T* occurs;
- there is a task on the ready queue for the active priority of *T* with a deadline earlier than the deadline of *T*; or
- there is a non-empty ready queue for that processor with a higher priority than the active priority of the running task.

In these cases, the currently running task is said to be preempted and is returned to the ready queue for its active priority.

For a task *T* to which policy EDF_Across_Priorities applies, the base priority is not a source of priority inheritance; the active priority when first activated or while it is blocked is defined as the maximum of the following:

- the lowest priority in the range specified as EDF_Across_Priorities that includes the base priority of *T*;
- the priorities, if any, currently inherited by *T*;
- the highest priority *P*, if any, less than the base priority of *T* such that one or more tasks are executing within a protected object with ceiling priority *P* and task *T* has an earlier deadline than all such tasks.

When a task *T* is first activated or becomes unblocked, it is added to the ready queue corresponding to this active priority. Until it becomes blocked again, the active priority of *T* remains no less than this value; it will exceed this value only while it is inheriting a higher priority.

When the setting of the base priority of a ready task takes effect and the new priority is in a range specified as EDF_Across_Priorities, the task is added to the ready queue corresponding to its new active priority, as determined above.

For all the operations defined in Dispatching.EDF, Tasking_Error is raised if the task identified by T has terminated. Program_Error is raised if the value of T is Null_Task_Id.

Bounded (Run-Time) Errors

If EDF_Across_Priorities is specified for priority range *Low*..*High*, it is a bounded error to declare a protected object with ceiling priority *Low* or to assign the value *Low* to attribute 'Priority. In either case either Program_Error is raised or the ceiling of the protected object is assigned the value *Low*+1.

Erroneous Execution

If a value of Task_Id is passed as a parameter to any of the subprograms of this package and the corresponding task object no longer exists, the execution of the program is erroneous.

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the deadline of a task to be delayed later than what is specified for a single processor.

NOTES

16 If two adjacent priority ranges, *A*..*B* and *B*+1..*C* are specified to have policy EDF_Across_Priorities then this is not equivalent to this policy being specified for the single range, *A*..*C*.

17 The above rules implement the preemption-level protocol (also called Stack Resource Policy protocol) for resource sharing under EDF dispatching. The preemption-level for a task is denoted by its base priority. The definition of a ceiling preemption-level for a protected object follows the existing rules for ceiling locking.

D.3 Priority Ceiling Locking

Replace paragraph 6:

A locking policy specifies the details of protected object locking. These rules specify whether or not protected objects have priorities, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking. The *locking policy* is specified by a Locking_Policy pragma. For implementation-defined locking policies, the effect of a Priority or Interrupt_Priority pragma on a protected object is implementation defined. If no Locking_Policy pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the effect of specifying either a Priority or Interrupt_Priority pragma for a protected object, are implementation defined.

by:

A locking policy specifies the details of protected object locking. All protected objects have a priority. The locking policy specifies the meaning of the priority of a protected object, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking. The *locking policy* is specified by a Locking_Policy pragma. For implementation-defined locking policies, the meaning of the priority of a protected object is implementation defined. If no Locking_Policy pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the meaning of the priority of a protected object, are implementation defined.

The expression of a Priority or Interrupt_Priority pragma (see D.1) is evaluated as part of the creation of the corresponding protected object and converted to the subtype System.Any_Priority or System.Interrupt_Priority, respectively. The value of the expression is the initial priority of the corresponding protected object. If no Priority or Interrupt_Priority pragma applies to a protected object, the initial priority is specified by the locking policy.

Replace paragraph 8:

- Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

by:

- Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1, or by assignment to the Priority attribute as described in D.5.2. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

Replace paragraph 9:

- The expression of a Priority or Interrupt_Priority pragma is evaluated as part of the creation of the corresponding protected object and converted to the subtype System.Any_Priority or System.Interrupt_Priority, respectively. The value of the expression is the ceiling priority of the corresponding protected object.

by:

- The initial ceiling priority of a protected object is equal to the initial priority for that object.

Replace paragraph 10:

- If an Interrupt_Handler or Attach_Handler pragma (see C.3.1) appears in a protected_definition without an Interrupt_Priority pragma, the ceiling priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt_Priority.

by:

- If an Interrupt_Handler or Attach_Handler pragma (see C.3.1) appears in a protected_definition without an Interrupt_Priority pragma, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt_Priority.

Replace paragraph 11:

- If no pragma Priority, Interrupt_Priority, Interrupt_Handler, or Attach_Handler is specified in the protected_definition, then the ceiling priority of the corresponding protected object is System.Priority'Last.

by:

- If no pragma Priority, Interrupt_Priority, Interrupt_Handler, or Attach_Handler is specified in the protected_definition, then the initial priority of the corresponding protected object is System.Priority'Last.

Insert after paragraph 13:

- When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails.

the new paragraphs:

Bounded (Run-Time) Errors

Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object. In this case one of the following applies:

- at any time prior to executing the entry body Program_Error is raised in the calling task;
- when the entry is open the entry body is executed at the ceiling priority of the protected object;
- when the entry is open the entry body is executed at the ceiling priority of the protected object and then Program_Error is raised in the calling task; or
- when the entry is open the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued.

Replace paragraph 15:

Implementations are allowed to define other locking policies, but need not support more than one such policy per partition.

by:

Implementations are allowed to define other locking policies, but need not support more than one locking policy per partition.

D.4 Entry Queuing Policies

Replace paragraph 7:

Two queuing policies, FIFO_Queuing and Priority_Queuing, are language defined. If no Queuing_Policy pragma appears in any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for this policy are specified in 9.5.3 and 9.7.1.

by:

Two queuing policies, FIFO_Queuing and Priority_Queuing, are language defined. If no Queuing_Policy pragma applies to any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for this policy are specified in 9.5.3 and 9.7.1.

Replace paragraph 15:

Implementations are allowed to define other queuing policies, but need not support more than one such policy per partition.

by:

Implementations are allowed to define other queuing policies, but need not support more than one queuing policy per partition.

Implementations are allowed to defer the reordering of entry queues following a change of base priority of a task blocked on the entry call if it is not practical to reorder the queue immediately.

D.5 Dynamic Priorities

Replace paragraph 1:

This clause specifies how the base priority of a task can be modified or queried at run time.

by:

This clause describes how the priority of an entity can be modified or queried at run time.

D.5.1 Dynamic Priorities for Tasks

[This changes the subclause of all of the existing text.]

Replace paragraph 3:

```
with System;
with Ada.Task_Identification; -- See C.7.1
package Ada.Dynamic_Priorities is
```

by:

```
with System;
with Ada.Task_Identification; -- See C.7.1
package Ada.Dynamic_Priorities is
  pragma Preelaborate(Dynamic_Priorities);
```

Replace paragraph 10:

Setting the task's base priority to the new value takes place as soon as is practical but not while the task is performing a protected action. This setting occurs no later then the next abort completion point of the task T (see 9.8).

by:

On a system with a single processor, the setting of the base priority of a task *T* to the new value occurs immediately at the first point when *T* is outside the execution of a protected action.

Delete paragraph 11:

If a task is blocked on a protected entry call, and the call is queued, it is a bounded error to raise its base priority above the ceiling priority of the corresponding protected object. When an entry call is cancelled, it is a bounded error if the priority of the calling task is higher than the ceiling priority of the corresponding protected object. In either of these cases, either Program_Error is raised in the task that called the entry, or its priority is temporarily lowered, or both, or neither.

Insert after paragraph 12:

If any subprogram in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous.

the new paragraph:

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor.

Replace paragraph 15:

29 Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the standard task dispatching policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged.

by:

29 Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the FIFO_Within_Priorities policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged.

D.5.2 Dynamic Priorities for Protected Objects

Insert new clause:

This clause specifies how the priority of a protected object can be modified or queried at run time.

Static Semantics

The following attribute is defined for a prefix P that denotes a protected object:

P'Priority
: Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

The initial value of this attribute is the initial value of the priority of the protected object, and can be changed by an assignment.

Dynamic Semantics

If the locking policy Ceiling_Locking (see D.3) is in effect then the ceiling priority of a protected object *P* is set to the value of *P*'Priority at the end of each protected action of *P*.

If the locking policy Ceiling_Locking is in effect, then for a protected object *P* with either an Attach_Handler or Interrupt_Handler pragma applying to one of its procedures, a check is made that the value to be assigned to *P*'Priority is in the range System.Interrupt_Priority. If the check fails, Program_Error is raised.

Metrics

The implementation shall document the following metric:

- The difference in execution time of calls to the following procedures in protected object P:

```
protected P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority);
   procedure Set_Ceiling (Pr : System.Any_Priority);
end P;

protected body P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority) is
   begin
      null;
   end;
   procedure Set_Ceiling (Pr : System.Any_Priority) is
   begin
      P'Priority := Pr;
   end;
end P;
```

NOTES

38 Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed.

D.7 Tasking Restrictions

Replace paragraph 4:

No_Nested_Finalization
: Objects with controlled, protected, or task parts and access types that designate such objects, shall be declared only at library level.

by:

No_Nested_Finalization
: Objects of a type that needs finalization (see 7.6) and access types that designate a type that needs finalization shall be declared only at library level.

Replace paragraph 9:

No_Dynamic_Priorities
: There are no semantic dependences on the package Dynamic_Priorities.

by:

No_Dynamic_Priorities
: There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority.

Replace paragraph 10:

No_Asynchronous_Control

There are no semantic dependences on the package Asynchronous_Task_Control.

by:

No_Dynamic_Attachment
: There is no call to any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

No_Local_Protected_Objects
: Protected objects shall be declared only at library level.

No_Local_Timing_Events
: Timing_Events shall be declared only at library level.

No_Protected_Type_Allocators
: There are no allocators for protected types or types containing protected type subcomponents.

No_Relative_Delay
: There are no delay_relative_statements.

No_Requeue_Statements
: There are no requeue_statements.

No_Select_Statements
: There are no select_statements.

No_Specific_Termination_Handlers
: There are no calls to the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

Simple_Barriers
: The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.

Replace paragraph 15:

This paragraph was deleted

by:

The following *restriction*_identifiers are language defined:

No_Task_Termination
: All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler (see C.7.3) set for the partition it should be called when the first task attempts to terminate.

Insert after paragraph 19:

Max_Tasks
: Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task. A value of zero prevents any task creation and, if a program contains a task creation, it is illegal. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

the new paragraph:

Max_Entry_Queue_Length
: Max_Entry_Queue_Length defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error at the point of the call or requeue.

D.8 Monotonic Time

Replace paragraph 14:

```
function Nanoseconds  (NS : Integer) return Time_Span;
function Microseconds (US : Integer) return Time_Span;
function Milliseconds (MS : Integer) return Time_Span;
```


by:

```
function Nanoseconds  (NS : Integer) return Time_Span;
function Microseconds (US : Integer) return Time_Span;
function Milliseconds (MS : Integer) return Time_Span;
function Seconds      (S  : Integer) return Time_Span;
function Minutes      (M  : Integer) return Time_Span;
```

Replace paragraph 24:

The function To_Duration converts the value TS to a value of type Duration. Similarly, the function To_Time_Span converts the value D to a value of type Time_Span. For both operations, the result is rounded to the nearest exactly representable value (away from zero if exactly halfway between two exactly representable values).

by:

The function To_Duration converts the value TS to a value of type Duration. Similarly, the function To_Time_Span converts the value D to a value of type Time_Span. For To_Duration, the result is rounded to the nearest value of type Duration (away from zero if exactly halfway between two values). If the result is outside the range of Duration, Constraint_Error is raised. For To_Time_Span, the value of D is first rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the value is converted to the type Time_Span.

Replace paragraph 26:

The functions Nanoseconds, Microseconds, and Milliseconds convert the input parameter to a value of the type Time_Span. NS, US, and MS are interpreted as a number of nanoseconds, microseconds, and milliseconds respectively. The result is rounded to the nearest exactly representable value (away from zero if exactly halfway between two exactly representable values).

by:

The functions Nanoseconds, Microseconds, Milliseconds, Seconds, and Minutes convert the input parameter to a value of the type Time_Span. NS, US, MS, S, and M are interpreted as a number of nanoseconds, microseconds, milliseconds, seconds, and minutes respectively. The input parameter is first converted to seconds and rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the rounded value is converted to the type Time_Span.

D.9 Delay Accuracy

Delete paragraph 14:

40 The execution time of a delay_statement that does not cause the task to be blocked (e.g. "**delay** 0.0;") is of interest in situations where delays are used to achieve voluntary round-robin task dispatching among equal-priority tasks.

D.10 Synchronous Task Control

Replace paragraph 3:

```
package Ada.Synchronous_Task_Control is
```

by:

```
package Ada.Synchronous_Task_Control is
   pragma Preelaborate(Synchronous_Task_Control);
```

D.11 Asynchronous Task Control

Replace paragraph 3:

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
   procedure Hold(T : in Ada.Task_Identification.Task_ID);
   procedure Continue(T : in Ada.Task_Identification.Task_ID);
```

```
      function Is_Held(T : Ada.Task_Identification.Task_ID)
         return Boolean;
   end Ada.Asynchronous_Task_Control;
```

by:

```
   with Ada.Task_Identification;
   package Ada.Asynchronous_Task_Control is
      pragma Preelaborate(Asynchronous_Task_Control);
      procedure Hold(T : in Ada.Task_Identification.Task_Id);
      procedure Continue(T : in Ada.Task_Identification.Task_Id);
      function Is_Held(T : Ada.Task_Identification.Task_Id)
         return Boolean;
   end Ada.Asynchronous_Task_Control;
```

Replace paragraph 4:

After the Hold operation has been applied to a task, the task becomes *held*. For each processor there is a conceptual *idle task*, which is always ready. The base priority of the idle task is below System.Any_Priority'First. The *held priority* is a constant of the type integer whose value is below the base priority of the idle task.

by:

After the Hold operation has been applied to a task, the task becomes *held*. For each processor there is a conceptual *idle task*, which is always ready. The base priority of the idle task is below System.Any_Priority'First. The *held priority* is a constant of the type Integer whose value is below the base priority of the idle task.

For any priority below System.Any_Priority'First, the task dispatching policy is FIFO_Within_Priorities.

Replace paragraph 5:

The Hold operation sets the state of T to held. For a held task: the task's own base priority does not constitute an inheritance source (see D.1), and the value of the held priority is defined to be such a source instead.

by:

The Hold operation sets the state of T to held. For a held task, the active priority is reevaluated as if the base priority of the task were the held priority.

Replace paragraph 6:

The Continue operation resets the state of T to not-held; T's active priority is then reevaluated as described in D.1. This time, T's base priority is taken into account.

by:

The Continue operation resets the state of T to not-held; its active priority is then reevaluated as determined by the task dispatching policy associated with its base priority.

D.13 Run-time Profiles

Insert new clause:

This clause specifies a mechanism for defining run-time profiles.

Syntax

The form of a pragma Profile is as follows:

pragma Profile (*profile*_identifier {, *profile*_pragma_argument_association});

Legality Rules

The *profile*_identifier shall be the name of a run-time profile. The semantics of any *profile*_pragma_argument_associations are defined by the run-time profile specified by the *profile*_identifier.

Static Semantics

A profile is equivalent to the set of configuration pragmas that is defined for each run-time profile.

Post-Compilation Rules

A pragma Profile is a configuration pragma. There may be more than one pragma Profile for a partition.

D.13.1 The Ravenscar Profile

Insert new clause:

This clause defines the Ravenscar profile.

Legality Rules

The *profile*_identifier Ravenscar is a run-time profile. For run-time profile Ravenscar, there shall be no *profile*_pragma_argument_associations.

Static Semantics

The run-time profile Ravenscar is equivalent to the following set of pragmas:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
               No_Abort_Statements,
               No_Dynamic_Attachment,
               No_Dynamic_Priorities,
               No_Implicit_Heap_Allocations,
               No_Local_Protected_Objects,
               No_Local_Timing_Events,
               No_Protected_Type_Allocators,
               No_Relative_Delay,
               No_Requeue_Statements,
               No_Select_Statements,
               No_Specific_Termination_Handlers,
               No_Task_Allocators,
               No_Task_Hierarchy,
               No_Task_Termination,
               Simple_Barriers,
               Max_Entry_Queue_Length => 1,
               Max_Protected_Entries => 1,
               Max_Task_Entries => 0,
               No_Dependence => Ada.Asynchronous_Task_Control,
               No_Dependence => Ada.Calendar,
               No_Dependence => Ada.Execution_Time.Group_Budget,
               No_Dependence => Ada.Execution_Time.Timers,
               No_Dependence => Ada.Task_Attributes);
```

NOTES

37 The effect of the Max_Entry_Queue_Length => 1 restriction applies only to protected entry queues due to the accompanying restriction of Max_Task_Entries => 0.

D.14 Execution Time

Insert new clause:

This clause describes a language-defined package to measure execution time.

Static Semantics

The following language-defined library package exists:

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is

   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
```

```
      CPU_Time_Unit  : constant := implementation-defined-real-number;
      CPU_Tick : constant Time_Span;

      function Clock
        (T : Ada.Task_Identification.Task_Id
             := Ada.Task_Identification.Current_Task)
        return CPU_Time;

      function "+"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
      function "+"  (Left : Time_Span; Right : CPU_Time) return CPU_Time;
      function "-"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
      function "-"  (Left : CPU_Time; Right : CPU_Time)  return Time_Span;

      function "<"  (Left, Right : CPU_Time) return Boolean;
      function "<=" (Left, Right : CPU_Time) return Boolean;
      function ">"  (Left, Right : CPU_Time) return Boolean;
      function ">=" (Left, Right : CPU_Time) return Boolean;

      procedure Split
        (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

      function Time_Of (SC : Seconds_Count;
                        TS : Time_Span := Time_Span_Zero) return CPU_Time;

   private
      ... --  not specified by the language
   end Ada.Execution_Time;
```

The *execution time* or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. It is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers and run-time services on behalf of the system.

The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers.

The CPU_Time value I represents the half-open execution-time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation-defined real number. For each task, the execution time value is set to zero at the creation of the task.

CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

Dynamic Semantics

CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A *CPU clock tick* is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals.

The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types.

The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 + TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit.

Erroneous Execution

For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous.

Implementation Requirements

The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task start-up to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

Documentation Requirements

The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

The implementation shall document the properties of the underlying mechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

Metrics

The implementation shall document the following metrics:

- An upper bound on the execution-time duration of a clock tick. This is a value D such that if t1 and t2 are any execution times of a given task such that t1 < t2 and Clock[t1] = Clock[t2] then t2-t1 <= D.
- An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution-time clock (as observed by calling the Clock function with the same Task_Id).
- An upper bound on the execution time of a call to the Clock function, in processor clock cycles.
- Upper bounds on the execution times of the operators of the type CPU_Time, in processor clock cycles.

Implementation Permissions

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the CPU_Time type.

Implementation Advice

When appropriate, implementations should provide configuration mechanisms to change the value of CPU_Tick.

D.14.1 Execution Time Timers

Insert new clause:

This clause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time.

Static Semantics

The following language-defined library package exists:

```
with System;
package Ada.Execution_Time.Timers is

   type Timer (T : not null access constant
                       Ada.Task_Identification.Task_Id) is
      tagged limited private;

   type Timer_Handler is
      access protected procedure (TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority :=
    implementation-defined;

   procedure Set_Handler (TM      : in out Timer;
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   function Current_Handler (TM : Timer) return Timer_Handler;
   procedure Cancel_Handler (TM        : in out Timer;
                             Cancelled :    out Boolean);

   function Time_Remaining (TM : Timer) return Time_Span;

   Timer_Resource_Error : exception;

private
   ... --  not specified by the language
```

```
end Ada.Execution_Time.Timers;
```

The type Timer represents an execution-time event for a single task and is capable of detecting execution-time overruns. The access discriminant T identifies the task concerned. The type Timer needs finalization (see 7.6).

An object of type Timer is said to be *set* if it is associated with a non-null value of type Timer_Handler and *cleared* otherwise. All Timer objects are initially cleared.

The type Timer_Handler identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a *handler*.

Dynamic Semantics

When a Timer object is created, or upon the first call of a Set_Handler procedure with the timer as parameter, the resources required to operate an execution-time timer based on the associated execution-time clock are allocated and initialized. If this operation would exceed the available resources, Timer_Resource_Error is raised.

The procedures Set_Handler associate the handler Handler with the timer TM; if Handler is **null**, the timer is cleared, otherwise it is set. The first procedure Set_Handler loads the timer TM with an interval specified by the Time_Span parameter. In this mode, the timer TM *expires* when the execution time of the task identified by TM.T.**all** has increased by In_Time; if In_Time is less than or equal to zero, the timer expires immediately. The second procedure Set_Handler loads the timer TM with the absolute value specified by At_Time. In this mode, the timer TM expires when the execution time of the task identified by TM.T.**all** reaches At_Time; if the value of At_Time has already been reached when Set_Handler is called, the timer expires immediately.

A call of a procedure Set_Handler for a timer that is already set replaces the handler and the (absolute or relative) execution time; if Handler is not **null**, the timer remains set.

When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

The function Current_Handler returns the handler associated with the timer TM if that timer is set; otherwise it returns **null**.

The procedure Cancel_Handler clears the timer if it is set. Cancelled is assigned True if the timer was set prior to it being cleared; otherwise it is assigned False.

The function Time_Remaining returns the execution time interval that remains until the timer TM would expire, if that timer is set; otherwise it returns Time_Span_Zero.

The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

As part of the finalization of an object of type Timer, the timer is cleared.

For all the subprograms defined in this package, Tasking_Error is raised if the task identified by TM.T.**all** has terminated, and Program_Error is raised if the value of TM.T.**all** is Task_Identification.Null_Task_Id.

An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

Erroneous Execution

For a call of any of the subprograms defined in this package, if the task identified by TM.T.**all** no longer exists, the execution of the program is erroneous.

Implementation Requirements

For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

When an object of type Timer is finalized, the system resources used by the timer shall be deallocated.

Implementation Permissions

Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded then Timer_Resource_Error is raised.

NOTES

46 A Timer_Handler can be associated with several Timer objects.

D.14.2 Group Execution Time Budgets

Insert new clause:

This clause describes a language-defined package to assign execution time budgets to groups of tasks.

Static Semantics

The following language-defined library package exists:

```
with System;
package Ada.Execution_Time.Group_Budgets is

  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access
       protected procedure (GB : in out Group_Budget);

  type Task_Array is array (Positive range <>) of
                                  Ada.Task_Identification.Task_Id;

  Min_Handler_Ceiling : constant System.Any_Priority :=
    implementation-defined;

  procedure Add_Task (GB : in out Group_Budget;
                      T  : in Ada.Task_Identification.Task_Id);
  procedure Remove_Task (GB: in out Group_Budget;
                         T  : in Ada.Task_Identification.Task_Id);
  function Is_Member (GB : Group_Budget;
                      T : Ada.Task_Identification.Task_Id) return Boolean;
  function Is_A_Group_Member
     (T : Ada.Task_Identification.Task_Id) return Boolean;
  function Members (GB : Group_Budget) return Task_Array;

  procedure Replenish (GB : in out Group_Budget; To : in Time_Span);
  procedure Add (GB : in out Group_Budget; Interval : in Time_Span);
  function Budget_Has_Expired (GB : Group_Budget) return Boolean;
  function Budget_Remaining (GB : Group_Budget) return Time_Span;

  procedure Set_Handler (GB      : in out Group_Budget;
                         Handler : in Group_Budget_Handler);
  function Current_Handler (GB : Group_Budget)
     return Group_Budget_Handler;
  procedure Cancel_Handler (GB        : in out Group_Budget;
                            Cancelled : out Boolean);

  Group_Budget_Error : exception;

private
    --  not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

The type Group_Budget represents an execution time budget to be used by a group of tasks. The type Group_Budget needs finalization (see 7.6). A task can belong to at most one group. Tasks of any priority can be added to a group.

An object of type Group_Budget has an associated nonnegative value of type Time_Span known as its *budget*, which is initially Time_Span_Zero. The type Group_Budget_Handler identifies a protected procedure to be executed by the implementation when the budget is *exhausted*, that is, reaches zero. Such a protected procedure is called a *handler*.

An object of type Group_Budget also includes a handler, which is a value of type Group_Budget_Handler. The handler of the object is said to be *set* if it is not null and *cleared* otherwise. The handler of all Group_Budget objects is initially cleared.

Dynamic Semantics

The procedure Add_Task adds the task identified by T to the group GB; if that task is already a member of some other group, Group_Budget_Error is raised.

The procedure Remove_Task removes the task identified by T from the group GB; if that task is not a member of the group GB, Group_Budget_Error is raised. After successful execution of this procedure, the task is no longer a member of any group.

The function Is_Member returns True if the task identified by T is a member of the group GB; otherwise it return False.

The function Is_A_Group_Member returns True if the task identified by T is a member of some group; otherwise it returns False.

The function Members returns an array of values of type Task_Identification.Task_Id identifying the members of the group GB. The order of the components of the array is unspecified.

The procedure Replenish loads the group budget GB with To as the Time_Span value. The exception Group_Budget_Error is raised if the Time_Span value To is non-positive. Any execution of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches Time_Span_Zero), the associated handler is executed if the handler of group budget GB is set. Nevertheless, the tasks continue to execute.

The procedure Add modifies the budget of the group GB. A positive value for Interval increases the budget. A negative value for Interval reduces the budget, but never below Time_Span_Zero. A zero value for Interval has no effect. A call of procedure Add that results in the value of the budget going to Time_Span_Zero causes the associated handler to be executed if the handler of the group budget GB is set.

The function Budget_Has_Expired returns True if the budget of group GB is exhausted (equal to Time_Span_Zero); otherwise it returns False.

The function Budget_Remaining returns the remaining budget for the group GB. If the budget is exhausted it returns Time_Span_Zero. This is the minimum value for a budget.

The procedure Set_Handler associates the handler Handler with the Group_Budget GB; if Handler is **null**, the handler of Group_Budget is cleared, otherwise it is set.

A call of Set_Handler for a Group_Budget that already has a handler set replaces the handler; if Handler is not **null**, the handler for Group_Budget remains set.

The function Current_Handler returns the handler associated with the group budget GB if the handler for that group budget is set; otherwise it returns **null**.

The procedure Cancel_Handler clears the handler for the group budget if it is set. Cancelled is assigned True if the handler for the group budget was set prior to it being cleared; otherwise it is assigned False.

The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

The precision of the accounting of task execution time to a Group_Budget is the same as that defined for execution-time clocks from the parent package.

As part of the finalization of an object of type Group_Budget all member tasks are removed from the group identified by that object.

If a task is a member of a Group_Budget when it terminates then as part of the finalization of the task it is removed from the group.

For all the operations defined in this package, Tasking_Error is raised if the task identified by T has terminated, and Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

An exception propagated from a handler invoked when the budget of a group of tasks becomes exhausted has no effect.

Erroneous Execution

For a call of any of the subprograms defined in this package, if the task identified by T no longer exists, the execution of the program is erroneous.

Implementation Requirements

For a given Group_Budget object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Group_Budget object. The replacement of a handler, by a call of Set_Handler, shall be performed atomically with respect to the execution of the handler.

NOTES

47 Clearing or setting of the handler of a group budget does not change the current value of the budget. Exhaustion or loading of a budget does not change whether the handler of the group budget is set or cleared.

48 A Group_Budget_Handler can be associated with several Group_Budget objects.

D.15 Timing Events

Insert new clause:

This clause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the need for a task or a delay statement.

Static Semantics

The following language-defined library package exists:

```
package Ada.Real_Time.Timing_Events is

  type Timing_Event is tagged limited private;
  type Timing_Event_Handler
       is access protected procedure (Event : in out Timing_Event);

  procedure Set_Handler (Event   : in out Timing_Event;
                         At_Time : in Time;
                         Handler : in Timing_Event_Handler);
  procedure Set_Handler (Event   : in out Timing_Event;
                         In_Time : in Time_Span;
                         Handler : in Timing_Event_Handler);
  function Current_Handler (Event : Timing_Event)
       return Timing_Event_Handler;
  procedure Cancel_Handler (Event     : in out Timing_Event;
                            Cancelled : out Boolean);

  function Time_Of_Event (Event : Timing_Event) return Time;

private
  ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event needs finalization (see 7.6).

An object of type Timing_Event is said to be *set* if it is associated with a non-null value of type Timing_Event_Handler and *cleared* otherwise. All Timing_Event objects are initially cleared.

The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a *handler*.

Dynamic Semantics

The procedures Set_Handler associate the handler Handler with the event Event; if Handler is **null**, the event is cleared, otherwise it is set. The first procedure Set_Handler sets the execution time for the event to be At_Time. The second procedure Set_Handler sets the execution time for the event to be Real_Time.Clock + In_Time.

A call of a procedure Set_Handler for an event that is already set replaces the handler and the time of execution; if Handler is not **null**, the event remains set.

As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event.

If the Ceiling_Locking policy (see D.3) is in effect when a procedure Set_Handler is called, a check is made that the ceiling priority of Handler.**all** is Interrupt_Priority'Last. If the check fails, Program_Error is raised.

If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past then the handler is executed immediately by the task executing the call of Set_Handler. The timing event Event is cleared.

The function Current_Handler returns the handler associated with the event Event if that event is set; otherwise it returns **null**.

The procedure Cancel_Handler clears the event if it is set. Cancelled is assigned True if the event was set prior to it being cleared; otherwise it is assigned False.

The function Time_Of_Event returns the time of the event if the event is set; otherwise it returns Real_Time.Time_First.

As part of the finalization of an object of type Timing_Event, the Timing_Event is cleared.

If several timing events are set for the same time, they are executed in FIFO order of being set.

An exception propagated from a handler invoked by a timing event has no effect.

Implementation Requirements

For a given Timing_Event object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timing_Event object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

Metrics

The implementation shall document the following metric:

- An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set.

Implementation Advice

The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.

NOTES

49 Since a call of Set_Handler is not a potentially blocking operation, it can be called from within a handler.

50 A Timing_Event_Handler can be associated with several Timing_Event objects.

Annex E: Distributed Systems

E.1 Partitions

Replace paragraph 10:

It is a bounded error for there to be cyclic elaboration dependences between the active partitions of a single distributed program. The possible effects are deadlock during elaboration, or the raising of Program_Error in one or all of the active partitions involved.

by:

It is a bounded error for there to be cyclic elaboration dependences between the active partitions of a single distributed program. The possible effects, in each of the partitions involved, are deadlock during elaboration, or the raising of Communication_Error or Program_Error.

E.2.2 Remote Types Library Units

Replace paragraph 8:

- if the full view of a type declared in the visible part of the library unit has a part that is of a non-remote access type, then that access type, or the type of some part that includes the access type subcomponent, shall have user-specified Read and Write attributes.

by:

- the full view of each type declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

Replace paragraph 11:

- A value of a remote access-to-subprogram type shall be converted only to another (subtype-conformant) remote access-to-subprogram type;

by:

- A value of a remote access-to-subprogram type shall be converted only to or from another (subtype-conformant) remote access-to-subprogram type;

Replace paragraph 14:

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall have either a nonlimited type or a type with Read and Write attributes specified via an attribute_definition_clause;

by:

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall support external streaming (see 13.13.2);

Replace paragraph 17:

- The Storage_Pool and Storage_Size attributes are not defined for remote access-to-class-wide types; the expected type for an allocator shall not be a remote access-to-class-wide type; a remote access-to-class-wide type shall not be an actual parameter for a generic formal access type.

by:

- The Storage_Pool attribute is not defined for a remote access-to-class-wide type; the expected type for an allocator shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The Storage_Size attribute of a remote access-to-class-wide type yields 0; it is not allowed in an attribute_definition_clause.

E.2.3 Remote Call Interface Library Units

Replace paragraph 14:

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter, or a formal parameter of a limited type unless that limited type has user-specified Read and Write attributes;

by:

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter or a parameter of a type that does not support external streaming (see 13.13.2);

E.5 Partition Communication Subsystem

Replace paragraph 1:

The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS. An implementation conforming to this Annex shall use the RPC interface to implement remote subprogram calls.

by:

The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS.

Insert after paragraph 27:

A body for the package System.RPC need not be supplied by the implementation.

the new paragraph:

An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this clause.

Annex F: Information Systems

Replace paragraph 4:

- the child packages Text_IO.Editing and Wide_Text_IO.Editing, which support formatted and localized output of decimal data, based on "picture String" values.

by:

- the child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing, which support formatted and localized output of decimal data, based on "picture String" values.

Replace paragraph 5:

See also: 3.5.9, "Fixed Point Types"; 3.5.10, "Operations of Fixed Point Types"; 4.6, "Type Conversions"; 13.3, "Operational and Representation Attributes"; A.10.9, "Input-Output for Real Types"; B.4, "Interfacing with COBOL"; B.3, "Interfacing with C and C++"; Annex G, "Numerics".

by:

See also: 3.5.9, "Fixed Point Types"; 3.5.10, "Operations of Fixed Point Types"; 4.6, "Type Conversions"; 13.3, "Operational and Representation Attributes"; A.10.9, "Input-Output for Real Types"; B.3, "Interfacing with C and C++"; B.4, "Interfacing with COBOL"; Annex G, "Numerics".

F.3 Edited Output for Decimal Types

Replace paragraph 1:

The child packages Text_IO.Editing and Wide_Text_IO.Editing provide localizable formatted text output, known as *edited output* , for decimal types. An edited output string is a function of a numeric value, program-specifiable locale elements, and a format control value. The numeric value is of some decimal type. The locale elements are:

by:

The child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing provide localizable formatted text output, known as *edited output*, for decimal types. An edited output string is a function of a numeric value, program-specifiable locale elements, and a format control value. The numeric value is of some decimal type. The locale elements are:

Replace paragraph 6:

For Text_IO.Editing the edited output and currency strings are of type String, and the locale characters are of type Character. For Wide_Text_IO.Editing their types are Wide_String and Wide_Character, respectively.

by:

For Text_IO.Editing the edited output and currency strings are of type String, and the locale characters are of type Character. For Wide_Text_IO.Editing their types are Wide_String and Wide_Character, respectively. For Wide_Wide_Text_IO.Editing their types are Wide_Wide_String and Wide_Wide_Character, respectively.

Replace paragraph 18:

An example of a picture String is "<###Z_ZZ9.99>". If the currency string is "FF", the separator character is ',', and the radix mark is '.' then the edited output string values for the decimal values 32.10 and –5432.10 are "bbFFbbb32.10b" and "(bFF5,432.10)", respectively, where 'b' indicates the space character.

by:

An example of a picture String is "<###Z_ZZ9.99>". If the currency string is "kr", the separator character is ',', and the radix mark is '.' then the edited output string values for the decimal values 32.10 and –5432.10 are "bbkrbbb32.10b" and "(bkr5,432.10)", respectively, where 'b' indicates the space character.

Replace paragraph 19:

The generic packages Text_IO.Decimal_IO and Wide_Text_IO.Decimal_IO (see A.10.9, "Input-Output for Real Types") provide text input and non-edited text output for decimal types.

by:

The generic packages Text_IO.Decimal_IO, Wide_Text_IO.Decimal_IO, and Wide_Wide_Text_IO.Decimal_IO (see A.10.9, "Input-Output for Real Types") provide text input and non-edited text output for decimal types.

Replace paragraph 20:

2 A picture String is of type Standard.String, both for Text_IO.Editing and Wide_Text_IO.Editing.

by:

2 A picture String is of type Standard.String, for all of Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing.

F.3.5 The Package Wide_Wide_Text_IO.Editing

Insert new clause:

Static Semantics

The child package Wide_Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that:

- each occurrence of Character is replaced by Wide_Wide_Character,
- each occurrence of Text_IO is replaced by Wide_Wide_Text_IO,
- the subtype of Default_Currency is Wide_Wide_String rather than String, and
- each occurrence of String in the generic package Decimal_Output is replaced by Wide_Wide_String.

NOTES

6 Each of the functions Wide_Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_Wide_String) as its parameter or result subtype, since a picture String is not localizable.

Annex G: Numerics

Replace paragraph 5:

- models of floating point and fixed point arithmetic on which the accuracy requirements of strict mode are based; and

by:

- models of floating point and fixed point arithmetic on which the accuracy requirements of strict mode are based;

Replace paragraph 6:

- the definitions of the model-oriented attributes of floating point types that apply in the strict mode.

by:

- the definitions of the model-oriented attributes of floating point types that apply in the strict mode; and
- features for the manipulation of real and complex vectors and matrices.

G.1.1 Complex Types

Replace paragraph 4:

```
type Imaginary is private;
```

by:

```
type Imaginary is private;
pragma Preelaborable_Initialization(Imaginary);
```

Replace paragraph 26:

Complex is a visible type with cartesian components.

by:

Complex is a visible type with Cartesian components.

Replace paragraph 56:

Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent; and reconverting to a cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

by:

Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent; and reconverting to a Cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

G.1.2 Complex Elementary Functions

Replace paragraph 2:

```
with Ada.Numerics.Generic_Complex_Types;
```

```
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions is
   pragma Pure(Generic_Complex_Elementary_Functions);
```

by:

```
with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions is
   pragma Pure(Generic_Complex_Elementary_Functions);
```

Replace paragraph 15:

The real (resp., imaginary) component of the result of the Arcsin and Arccos (resp., Arctanh) functions is discontinuous as the parameter X crosses the real axis to the left of –1.0 or the right of 1.0.

by:

The imaginary component of the result of the Arcsin, Arccos, and Arctanh functions is discontinuous as the parameter X crosses the real axis to the left of –1.0 or the right of 1.0.

Replace paragraph 16:

The real (resp., imaginary) component of the result of the Arctan (resp., Arcsinh) function is discontinuous as the parameter X crosses the imaginary axis below *–i* or above *i*.

by:

The real component of the result of the Arctan and Arcsinh functions is discontinuous as the parameter X crosses the imaginary axis below *–i* or above *i*.

Replace paragraph 17:

The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis between *–i* and *i*.

by:

The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis below *–i* or above *i*.

Replace paragraph 20:

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

by:

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in Numerics.Generic_Elementary_Functions. (For Arctan and Arccot, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.)

Replace paragraph 41:

- the Log function yields an imaginary result; and the Arcsin and Arccos functions yield a real result.

by:

- When the parameter X has the value –1.0, the Log function yields an imaginary result; and the Arcsin and Arccos functions yield a real result.

G.1.3 Complex Input-Output

Insert before paragraph 10:

The semantics of the Get and Put procedures are as follows:

the new paragraph:

The library package Complex_Text_IO defines the same subprograms as Text_IO.Complex_IO, except that the predefined type Float is systematically substituted for Real, and the type Numerics.Complex_Types.Complex is systematically substituted for Complex throughout. Non-generic equivalents of Text_IO.Complex_IO corresponding to each of the other predefined floating point types are defined similarly, with the names Short_Complex_Text_IO, Long_Complex_Text_IO, etc.

Replace paragraph 28:

Reads a complex value from the beginning of the given string, following the same rule as the Get procedure that reads a complex value from a file, but treating the end of the string as a line terminator. Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence. Returns in Last the index value such that From(Last) is the last character read.

by:

Reads a complex value from the beginning of the given string, following the same rule as the Get procedure that reads a complex value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence. Returns in Last the index value such that From(Last) is the last character read.

G.1.5 The Package Wide_Wide_Text_IO.Complex_IO

Insert new clause:

Static Semantics

Implementations shall also provide the generic library package Wide_Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Wide_Text_IO and String by Wide_Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide wide characters.

G.2.2 Model-Oriented Attributes of Floating Point Types

Replace paragraph 3:

Yields the number of digits in the mantissa of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to *Ceiling*(*d* * log(10) / log(*T*'Machine_Radix)) + 1, where *d* is the requested decimal precision of *T*. In addition, it shall be less than or equal to the value of *T*'Machine_Mantissa. This attribute yields a value of the type *universal_integer*.

by:

Yields the number of digits in the mantissa of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to

```
Ceiling(d * log(10) / log(T'Machine_Radix)) + g
```

where *d* is the requested decimal precision of *T*, and *g* is 0 if *T*'Machine_Radix is a positive power of 10 and 1 otherwise. In addition, *T*'Model_Mantissa shall be less than or equal to the value of *T*'Machine_Mantissa. This attribute yields a value of the type *universal_integer*.

G.2.4 Accuracy Requirements for the Elementary Functions

Replace paragraph 11:

The prescribed results specified in A.5.1 for certain functions at particular parameter values take precedence over the maximum relative error bounds; effectively, they narrow to a single value the result interval allowed by the maximum relative error bounds. Additional rules with a similar effect are given by the table below for the inverse trigonometric functions, at particular parameter values for which the mathematical result is possibly not a model number of *EF*.Float_Type (or is, indeed, even transcendental). In each table entry, the values of the parameters are such that the result lies on the axis between two quadrants; the corresponding accuracy rule, which takes precedence over the maximum relative error bounds, is that the result interval is the model interval of *EF*.Float_Type associated with the exact mathematical result given in the table.

by:

The prescribed results specified in A.5.1 for certain functions at particular parameter values take precedence over the maximum relative error bounds; effectively, they narrow to a single value the result interval allowed by the maximum relative error bounds. Additional rules with a similar effect are given by table G-1 for the inverse trigonometric functions, at particular parameter values for which the mathematical result is possibly not a model number of *EF*.Float_Type (or is, indeed, even transcendental). In each table entry, the values of the parameters are such that the result lies on the axis between two quadrants; the corresponding accuracy rule, which takes precedence over the maximum relative error bounds, is that the result interval is the model interval of *EF*.Float_Type associated with the exact mathematical result given in the table.

G.2.6 Accuracy Requirements for Complex Arithmetic

Replace paragraph 6:

The error bounds for particular complex functions are tabulated below. In the table, the error bound is given as the coefficient of *CT*.Real'Model_Epsilon.

by:

The error bounds for particular complex functions are tabulated in table G-2. In the table, the error bound is given as the coefficient of *CT*.Real'Model_Epsilon.

Replace paragraph 13:

The amount by which a component of the result of an inverse trigonometric or inverse hyperbolic function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in G.1.2, is limited. The rule is that the result belongs to the smallest model interval of *CT*.Real that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence to the maximum error bounds, effectively narrowing the result interval allowed by them.

by:

The amount by which a component of the result of an inverse trigonometric or inverse hyperbolic function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in G.1.2, is limited. The rule is that the result belongs to the smallest model interval of *CT*.Real that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum error bounds, effectively narrowing the result interval allowed by them.

G.3 Vector and Matrix Manipulation

Insert new clause:

Types and operations for the manipulation of real vectors and matrices are provided in Generic_Real_Arrays, which is defined in G.3.1. Types and operations for the manipulation of complex vectors and matrices are provided in Generic_Complex_Arrays, which is defined in G.3.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these packages for each of the predefined floating point types are also provided as children of Numerics.

G.3.1 Real Vectors and Matrices

Insert new clause:

Static Semantics

The generic library package Numerics.Generic_Real_Arrays has the following declaration:

```
generic
   type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays is
   pragma Pure(Generic_Real_Arrays);

   -- Types

   type Real_Vector is array (Integer range <>) of Real'Base;
   type Real_Matrix is array (Integer range <>, Integer range <>)
                                    of Real'Base;

   -- Subprograms for Real_Vector types

   -- Real_Vector arithmetic operations

   function "+"   (Right : Real_Vector)       return Real_Vector;
   function "-"   (Right : Real_Vector)       return Real_Vector;
   function "abs" (Right : Real_Vector)       return Real_Vector;

   function "+"   (Left, Right : Real_Vector) return Real_Vector;
   function "-"   (Left, Right : Real_Vector) return Real_Vector;

   function "*"   (Left, Right : Real_Vector) return Real'Base;

   function "abs" (Right : Real_Vector)       return Real'Base;

   -- Real_Vector scaling operations

   function "*" (Left : Real'Base;   Right : Real_Vector)
      return Real_Vector;
   function "*" (Left : Real_Vector; Right : Real'Base)
      return Real_Vector;
   function "/" (Left : Real_Vector; Right : Real'Base)
      return Real_Vector;

   -- Other Real_Vector operations

   function Unit_Vector (Index : Integer;
                         Order : Positive;
                         First : Integer := 1) return Real_Vector;

   -- Subprograms for Real_Matrix types

   -- Real_Matrix arithmetic operations

   function "+"       (Right : Real_Matrix) return Real_Matrix;
   function "-"       (Right : Real_Matrix) return Real_Matrix;
   function "abs"     (Right : Real_Matrix) return Real_Matrix;
   function Transpose (X     : Real_Matrix) return Real_Matrix;

   function "+" (Left, Right : Real_Matrix) return Real_Matrix;
   function "-" (Left, Right : Real_Matrix) return Real_Matrix;
   function "*" (Left, Right : Real_Matrix) return Real_Matrix;

   function "*" (Left, Right : Real_Vector) return Real_Matrix;

   function "*" (Left : Real_Vector; Right : Real_Matrix)
      return Real_Vector;
   function "*" (Left : Real_Matrix; Right : Real_Vector)
```

```
      return Real_Vector;

   -- Real_Matrix scaling operations

   function "*" (Left : Real'Base;   Right : Real_Matrix)
      return Real_Matrix;
   function "*" (Left : Real_Matrix; Right : Real'Base)
      return Real_Matrix;
   function "/" (Left : Real_Matrix; Right : Real'Base)
      return Real_Matrix;

   -- Real_Matrix inversion and related operations

   function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
   function Solve (A, X : Real_Matrix) return Real_Matrix;
   function Inverse (A : Real_Matrix) return Real_Matrix;
   function Determinant (A : Real_Matrix) return Real'Base;

   -- Eigenvalues and vectors of a real symmetric matrix

   function Eigenvalues(A : Real_Matrix) return Real_Vector;

   procedure Eigensystem(A       : in  Real_Matrix;
                         Values  : out Real_Vector;
                         Vectors : out Real_Matrix);

   -- Other Real_Matrix operations

   function Unit_Matrix (Order            : Positive;
                         First_1, First_2 : Integer := 1)
                                            return Real_Matrix;

end Ada.Numerics.Generic_Real_Arrays;
```

The library package Numerics.Real_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Real_Arrays, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Real_Arrays, Numerics.Long_Real_Arrays, etc.

Two types are defined and exported by Numerics.Generic_Real_Arrays. The composite type Real_Vector is provided to represent a vector with components of type Real; it is defined as an unconstrained, one-dimensional array with an index of type Integer. The composite type Real_Matrix is provided to represent a matrix with components of type Real; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

The effect of the various subprograms is as described below. In most cases the subprograms are described in terms of corresponding scalar operations of the type Real; any exception raised by those operations is propagated by the array operation. Moreover, the accuracy of the result for each individual component is as defined for the scalar operation unless stated otherwise.

In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result is outside the range of Real'Base even though the mathematical final result would not be.

```
function "+"   (Right : Real_Vector) return Real_Vector;
function "-"   (Right : Real_Vector) return Real_Vector;
function "abs" (Right : Real_Vector) return Real_Vector;
```

Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index range of the result is Right'Range.

```
function "+" (Left, Right : Real_Vector) return Real_Vector;
function "-" (Left, Right : Real_Vector) return Real_Vector;
```

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

```
function "*" (Left, Right : Real_Vector) return Real'Base;
```


This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

```
function "abs" (Right : Real_Vector) return Real'Base;
```

This operation returns the L2-norm of Right (the square root of the inner product of the vector with itself).

```
function "*" (Left : Real'Base; Right : Real_Vector) return Real_Vector;
```

This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index range of the result is Right'Range.

```
function "*" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
function "/" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
```

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index range of the result is Left'Range.

```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Real_Vector;
```

This function returns a *unit vector* with Order components and a lower bound of First. All components are set to 0.0 except for the Index component which is set to 1.0. Constraint_Error is raised if Index < First, Index > First + Order – 1 or if First + Order – 1 > Integer'Last.

```
function "+"   (Right : Real_Matrix) return Real_Matrix;
function "-"   (Right : Real_Matrix) return Real_Matrix;
function "abs" (Right : Real_Matrix) return Real_Matrix;
```

Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index ranges of the result are those of Right.

```
function Transpose (X : Real_Matrix) return Real_Matrix;
```

This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

```
function "+" (Left, Right : Real_Matrix) return Real_Matrix;
function "-" (Left, Right : Real_Matrix) return Real_Matrix;
```

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

```
function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```

This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

```
function "*" (Left, Right : Real_Vector) return Real_Matrix;
```

This operation returns the outer product of a (column) vector Left by a (row) vector Right using the operation "*" of the type Real for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

```
function "*" (Left : Real_Vector; Right : Real_Matrix) return Real_Vector;
```

This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

```
function "*" (Left : Real_Matrix; Right : Real_Vector) return Real_Vector;
```

This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

```
function "*" (Left : Real'Base; Right : Real_Matrix) return Real_Matrix;
```

This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index ranges of the result are those of Right.

```
function "*" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;
function "/" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;
```

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index ranges of the result are those of Left.

```
function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
```

This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Solve (A, X : Real_Matrix) return Real_Matrix;
```

This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Inverse (A : Real_Matrix) return Real_Matrix;
```

This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Determinant (A : Real_Matrix) return Real'Base;
```

This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).

```
function Eigenvalues(A : Real_Matrix) return Real_Vector;
```

This function returns the eigenvalues of the symmetric matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not symmetric.

```
procedure Eigensystem(A       : in  Real_Matrix;
                      Values  : out Real_Vector;
                      Vectors : out Real_Matrix);
```

This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index ranges of the parameter Vectors are those of A. Argument_Error is raised if the matrix A is not symmetric.

```
function Unit_Matrix (Order           : Positive;
                      First_1, First_2 : Integer := 1) return Real_Matrix;
```

This function returns a square *unit matrix* with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to 0.0 except for the main diagonal, whose components are set to 1.0. Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.

Implementation Requirements

Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real in both the strict mode and the relaxed mode (see G.2).

For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as

g = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)

For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* is defined as above.

Documentation Requirements

Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

Implementation Permissions

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

Implementation Advice

Implementations should implement the Solve and Inverse functions using established techniques such as LU decomposition with row interchanges followed by back and forward substitution. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done then it should be documented.

It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient.

The test that a matrix is symmetric should be performed by using the equality operator to compare the relevant components.

G.3.2 Complex Vectors and Matrices

Insert new clause:

Static Semantics

The generic library package Numerics.Generic_Complex_Arrays has the following declaration:

```
with Ada.Numerics.Generic_Real_Arrays, Ada.Numerics.Generic_Complex_Types;
generic
   with package Real_Arrays   is new
      Ada.Numerics.Generic_Real_Arrays   (<>);
   use Real_Arrays;
   with package Complex_Types is new
      Ada.Numerics.Generic_Complex_Types (Real);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays is
   pragma Pure(Generic_Complex_Arrays);

   -- Types

   type Complex_Vector is array (Integer range <>) of Complex;
   type Complex_Matrix is array (Integer range <>,
                                 Integer range <>) of Complex;

   -- Subprograms for Complex_Vector types

   -- Complex_Vector selection, conversion and composition operations

   function Re (X : Complex_Vector) return Real_Vector;
   function Im (X : Complex_Vector) return Real_Vector;

   procedure Set_Re (X  : in out Complex_Vector;
                     Re : in     Real_Vector);
   procedure Set_Im (X  : in out Complex_Vector;
                     Im : in     Real_Vector);

   function Compose_From_Cartesian (Re     : Real_Vector)
```

```
   return Complex_Vector;
function Compose_From_Cartesian (Re, Im : Real_Vector)
   return Complex_Vector;

function Modulus  (X     : Complex_Vector) return Real_Vector;
function "abs"    (Right : Complex_Vector) return Real_Vector
                                              renames Modulus;
function Argument (X     : Complex_Vector) return Real_Vector;
function Argument (X     : Complex_Vector;
                   Cycle : Real'Base)      return Real_Vector;

function Compose_From_Polar (Modulus, Argument : Real_Vector)
   return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
   return Complex_Vector;

-- Complex_Vector arithmetic operations

function "+"       (Right  : Complex_Vector) return Complex_Vector;
function "-"       (Right  : Complex_Vector) return Complex_Vector;
function Conjugate (X      : Complex_Vector) return Complex_Vector;

function "+"  (Left, Right : Complex_Vector) return Complex_Vector;
function "-"  (Left, Right : Complex_Vector) return Complex_Vector;

function "*"  (Left, Right : Complex_Vector) return Complex;

function "abs"     (Right : Complex_Vector) return Complex;

-- Mixed Real_Vector and Complex_Vector arithmetic operations

function "+" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;

function "*" (Left  : Real_Vector;    Right : Complex_Vector)
   return Complex;
function "*" (Left  : Complex_Vector; Right : Real_Vector)
   return Complex;

-- Complex_Vector scaling operations

function "*" (Left  : Complex;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Complex)        return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Complex)        return Complex_Vector;

function "*" (Left  : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;

-- Other Complex_Vector operations

function Unit_Vector (Index : Integer;
                      Order : Positive;
```

```
                     First : Integer := 1) return Complex_Vector;

-- Subprograms for Complex_Matrix types

-- Complex_Matrix selection, conversion and composition operations

function Re (X : Complex_Matrix) return Real_Matrix;
function Im (X : Complex_Matrix) return Real_Matrix;

procedure Set_Re (X  : in out Complex_Matrix;
                  Re : in     Real_Matrix);
procedure Set_Im (X  : in out Complex_Matrix;
                  Im : in     Real_Matrix);

function Compose_From_Cartesian (Re     : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;

function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                             renames Modulus;

function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;

function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;

-- Complex_Matrix arithmetic operations

function "+"       (Right : Complex_Matrix) return Complex_Matrix;
function "-"       (Right : Complex_Matrix) return Complex_Matrix;
function Conjugate (X     : Complex_Matrix) return Complex_Matrix;
function Transpose (X     : Complex_Matrix) return Complex_Matrix;

function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;

function "*" (Left, Right : Complex_Vector) return Complex_Matrix;

function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;

-- Mixed Real_Matrix and Complex_Matrix arithmetic operations

function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

```
      function "*" (Left  : Real_Vector;
                    Right : Complex_Vector) return Complex_Matrix;
      function "*" (Left  : Complex_Vector;
                    Right : Real_Vector)    return Complex_Matrix;

      function "*" (Left  : Real_Vector;
                    Right : Complex_Matrix) return Complex_Vector;
      function "*" (Left  : Complex_Vector;
                    Right : Real_Matrix)    return Complex_Vector;
      function "*" (Left  : Real_Matrix;
                    Right : Complex_Vector) return Complex_Vector;
      function "*" (Left  : Complex_Matrix;
                    Right : Real_Vector)    return Complex_Vector;

      -- Complex_Matrix scaling operations

      function "*" (Left  : Complex;
                    Right : Complex_Matrix) return Complex_Matrix;
      function "*" (Left  : Complex_Matrix;
                    Right : Complex)        return Complex_Matrix;
      function "/" (Left  : Complex_Matrix;
                    Right : Complex)        return Complex_Matrix;

      function "*" (Left  : Real'Base;
                    Right : Complex_Matrix) return Complex_Matrix;
      function "*" (Left  : Complex_Matrix;
                    Right : Real'Base)      return Complex_Matrix;
      function "/" (Left  : Complex_Matrix;
                    Right : Real'Base)      return Complex_Matrix;

      -- Complex_Matrix inversion and related operations

      function Solve (A : Complex_Matrix; X : Complex_Vector)
         return Complex_Vector;
      function Solve (A, X : Complex_Matrix) return Complex_Matrix;
      function Inverse (A : Complex_Matrix) return Complex_Matrix;
      function Determinant (A : Complex_Matrix) return Complex;

      -- Eigenvalues and vectors of a Hermitian matrix

      function Eigenvalues(A : Complex_Matrix) return Real_Vector;

      procedure Eigensystem(A       : in  Complex_Matrix;
                            Values  : out Real_Vector;
                            Vectors : out Complex_Matrix);

      -- Other Complex_Matrix operations

      function Unit_Matrix (Order            : Positive;
                            First_1, First_2 : Integer := 1)
                              return Complex_Matrix;

   end Ada.Numerics.Generic_Complex_Arrays;
```

The library package Numerics.Complex_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Complex_Arrays, except that the predefined type Float is systematically substituted for Real'Base, and the Real_Vector and Real_Matrix types exported by Numerics.Real_Arrays are systematically substituted for Real_Vector and Real_Matrix, and the Complex type exported by Numerics.Complex_Types is systematically substituted for Complex, throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Arrays, Numerics.Long_Complex_Arrays, etc.

Two types are defined and exported by Numerics.Generic_Complex_Arrays. The composite type Complex_Vector is provided to represent a vector with components of type Complex; it is defined as an unconstrained one-dimensional array with an index of type Integer. The composite type Complex_Matrix is provided to represent a matrix with components of type Complex; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

The effect of the various subprograms is as described below. In many cases they are described in terms of corresponding scalar operations in Numerics.Generic_Complex_Types. Any exception raised by those operations is propagated by the array subprogram. Moreover, any constraints on the parameters and the accuracy of the result for each individual component are as defined for the scalar operation.

In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result has a component outside the range of Real'Base even though the final mathematical result would not.

```
function Re (X : Complex_Vector) return Real_Vector;
function Im (X : Complex_Vector) return Real_Vector;
```

Each function returns a vector of the specified Cartesian components of X. The index range of the result is X'Range.

```
procedure Set_Re (X  : in out Complex_Vector; Re : in Real_Vector);
procedure Set_Im (X  : in out Complex_Vector; Im : in Real_Vector);
```

Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length is not equal to Re'Length or Im'Length.

```
function Compose_From_Cartesian (Re     : Real_Vector) return Complex_Vector;
function Compose_From_Cartesian (Re, Im : Real_Vector) return Complex_Vector;
```

Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index range of the result is Re'Range. Constraint_Error is raised if Re'Length is not equal to Im'Length.

```
function Modulus  (X     : Complex_Vector) return Real_Vector;
function "abs"    (Right : Complex_Vector) return Real_Vector
                                              renames Modulus;
function Argument (X     : Complex_Vector) return Real_Vector;
function Argument (X     : Complex_Vector;
                   Cycle : Real'Base)      return Real_Vector;
```

Each function calculates and returns a vector of the specified polar components of X or Right using the corresponding function in Numerics.Generic_Complex_Types. The index range of the result is X'Range or Right'Range.

```
function Compose_From_Polar (Modulus, Argument : Real_Vector)
   return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
   return Complex_Vector;
```

Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of polar components using the corresponding function in Numerics.Generic_Complex_Types on matching components of Modulus and Argument. The index range of the result is Modulus'Range. Constraint_Error is raised if Modulus'Length is not equal to Argument'Length.

```
function "+" (Right : Complex_Vector) return Complex_Vector;
function "-" (Right : Complex_Vector) return Complex_Vector;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of Right. The index range of the result is Right'Range.

```
function Conjugate (X : Complex_Vector) return Complex_Vector;
```

This function returns the result of applying the appropriate function Conjugate in Numerics.Generic_Complex_Types to each component of X. The index range of the result is X'Range.

```
function "+" (Left, Right : Complex_Vector) return Complex_Vector;
function "-" (Left, Right : Complex_Vector) return Complex_Vector;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

```
function "*" (Left, Right : Complex_Vector) return Complex;
```

This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

```
function "abs" (Right : Complex_Vector) return Complex;
```

This operation returns the Hermitian L2-norm of Right (the square root of the inner product of the vector with its conjugate).

```
function "+" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

```
function "*" (Left : Real_Vector;    Right : Complex_Vector) return Complex;
function "*" (Left : Complex_Vector; Right : Real_Vector)    return Complex;
```

Each operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. These operations involve an inner product.

```
function "*" (Left : Complex; Right : Complex_Vector) return Complex_Vector;
```

This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in Numerics.Generic_Complex_Types. The index range of the result is Right'Range.

```
function "*" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
function "/" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of the vector Left and the complex number Right. The index range of the result is Left'Range.

```
function "*" (Left : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
```

This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in Numerics.Generic_Complex_Types. The index range of the result is Right'Range.

```
function "*" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
function "/" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of the vector Left and the real number Right. The index range of the result is Left'Range.

```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;
```

This function returns a *unit vector* with Order components and a lower bound of First. All components are set to (0.0, 0.0) except for the Index component which is set to (1.0, 0.0). Constraint_Error is raised if Index < First, Index > First + Order – 1, or if First + Order – 1 > Integer'Last.

```
function Re (X : Complex_Matrix) return Real_Matrix;
function Im (X : Complex_Matrix) return Real_Matrix;
```

Each function returns a matrix of the specified Cartesian components of X. The index ranges of the result are those of X.

```
procedure Set_Re (X : in out Complex_Matrix; Re : in Real_Matrix);
procedure Set_Im (X : in out Complex_Matrix; Im : in Real_Matrix);
```

Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length(1) is not equal to Re'Length(1) or Im'Length(1) or if X'Length(2) is not equal to Re'Length(2) or Im'Length(2).

```
function Compose_From_Cartesian (Re     : Real_Matrix) return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix) return Complex_Matrix;
```

Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index ranges of the result are those of Re. Constraint_Error is raised if Re'Length(1) is not equal to Im'Length(1) or Re'Length(2) is not equal to Im'Length(2).

```
function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                             renames Modulus;
function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;
```

Each function calculates and returns a matrix of the specified polar components of X or Right using the corresponding function in Numerics.Generic_Complex_Types. The index ranges of the result are those of X or Right.

```
function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;
```

Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of polar components using the corresponding function in Numerics.Generic_Complex_Types on matching components of Modulus and Argument. The index ranges of the result are those of Modulus. Constraint_Error is raised if Modulus'Length(1) is not equal to Argument'Length(1) or Modulus'Length(2) is not equal to Argument'Length(2).

```
function "+" (Right : Complex_Matrix) return Complex_Matrix;
function "-" (Right : Complex_Matrix) return Complex_Matrix;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of Right. The index ranges of the result are those of Right.

```
function Conjugate (X : Complex_Matrix) return Complex_Matrix;
```

This function returns the result of applying the appropriate function Conjugate in Numerics.Generic_Complex_Types to each component of X. The index ranges of the result are those of X.

```
function Transpose (X : Complex_Matrix) return Complex_Matrix;
```

This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

```
function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

```
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

```
function "*" (Left, Right : Complex_Vector) return Complex_Matrix;
```

This operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in Numerics.Generic_Complex_Types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

```
function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
```

This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

```
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;
```

This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

```
function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

```
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

Each operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). These operations involve inner products.

```
function "*" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Matrix;
function "*" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Matrix;
```

Each operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in Numerics.Generic_Complex_Types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

```
function "*" (Left  : Real_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real_Matrix)    return Complex_Vector;
```

Each operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). These operations involve inner products.

```
function "*" (Left  : Real_Matrix;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Real_Vector)    return Complex_Vector;
```

Each operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. These operations involve inner products.

```
function "*" (Left : Complex; Right : Complex_Matrix) return Complex_Matrix;
```

This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in Numerics.Generic_Complex_Types. The index ranges of the result are those of Right.

```
function "*" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
function "/" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of the matrix Left and the complex number Right. The index ranges of the result are those of Left.

```
function "*" (Left : Real'Base;
              Right : Complex_Matrix) return Complex_Matrix;
```

This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in Numerics.Generic_Complex_Types. The index ranges of the result are those of Right.

```
function "*" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
function "/" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
```

Each operation returns the result of applying the corresponding operation in Numerics.Generic_Complex_Types to each component of the matrix Left and the real number Right. The index ranges of the result are those of Left.

```
function Solve (A : Complex_Matrix; X : Complex_Vector) return Complex_Vector;
```

This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Solve (A, X : Complex_Matrix) return Complex_Matrix;
```

This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Inverse (A : Complex_Matrix) return Complex_Matrix;
```

This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Determinant (A : Complex_Matrix) return Complex;
```

This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).

```
function Eigenvalues(A : Complex_Matrix) return Real_Vector;
```

This function returns the eigenvalues of the Hermitian matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not Hermitian.

```
procedure Eigensystem(A       : in  Complex_Matrix;
                      Values  :  out Real_Vector;
                      Vectors :  out Complex_Matrix);
```

This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues.

Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index ranges of the parameter Vectors are those of A. Argument_Error is raised if the matrix A is not Hermitian.

```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1)
                                         return Complex_Matrix;
```

This function returns a square *unit matrix* with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to (0.0, 0.0) except for the main diagonal, whose components are set to (1.0, 0.0). Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.

Implementation Requirements

Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real'Base and Complex in both the strict mode and the relaxed mode (see G.2).

For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as

g = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)

for mixed complex and real operands

g = sqrt(2.0) * *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)

for two complex operands

For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* has the definition appropriate for two complex operands.

Documentation Requirements

Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

Implementation Permissions

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

Although many operations are defined in terms of operations from Numerics.Generic_Complex_Types, they need not be implemented by calling those operations provided that the effect is the same.

Implementation Advice

Implementations should implement the Solve and Inverse functions using established techniques. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done then it should be documented.

It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient.

The test that a matrix is Hermitian should use the equality operator to compare the real components and negation followed by equality to compare the imaginary components (see G.2.1).

Implementations should not perform operations on mixed complex and real operands by first converting the real operand to complex. See G.1.1.

Annex H: Safety and Security

Replace the title:

Safety and Security

by:

High Integrity Systems

Replace paragraph 1:

This Annex addresses requirements for systems that are safety critical or have security constraints. It provides facilities and specifies documentation requirements that relate to several needs:

by:

This Annex addresses requirements for high integrity systems (including safety-critical systems and security-critical systems). It provides facilities and specifies documentation requirements that relate to several needs:

H.1 Pragma Normalize_Scalars

Replace paragraph 5:

If a pragma Normalize_Scalars applies, the implementation shall document the implicit initial value for scalar subtypes, and shall identify each case in which such a value is used and is not an invalid representation.

by:

If a pragma Normalize_Scalars applies, the implementation shall document the implicit initial values for scalar subtypes, and shall identify each case in which such a value is used and is not an invalid representation.

Replace paragraph 6:

Whenever possible, the implicit initial value for a scalar subtype should be an invalid representation (see 13.9.1).

by:

Whenever possible, the implicit initial values for a scalar subtype should be an invalid representation (see 13.9.1).

H.3.1 Pragma Reviewable

Replace paragraph 8:

- For each reference to a scalar object, an identification of the reference as either "known to be initialized," or "possibly uninitialized," independent of whether pragma Normalize_Scalars applies;

by:

- For each read of a scalar object, an identification of the read as either "known to be initialized," or "possibly uninitialized," independent of whether pragma Normalize_Scalars applies;

H.3.2 Pragma Inspection_Point

Replace paragraph 5:

An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments and the object is visible at the inspection point.

by:

An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments and the declaration of the object is visible at the inspection point.

Replace paragraph 9:

7 The implementation is not allowed to perform "dead store elimination" on the last assignment to a variable prior to a point where the variable is inspectable. Thus an inspection point has the effect of an implicit reference to each of its inspectable objects.

by:

7 The implementation is not allowed to perform "dead store elimination" on the last assignment to a variable prior to a point where the variable is inspectable. Thus an inspection point has the effect of an implicit read of each of its inspectable objects.

H.4 Safety and Security Restrictions

Replace the title:

Safety and Security Restrictions

by:

High Integrity Restrictions

Delete paragraph 2:

The following restrictions, the same as in D.7, apply in this Annex: No_Task_Hierarchy, No_Abort_Statement, No_Implicit_Heap_Allocation, Max_Task_Entries is 0, Max_Asynchronous_Select_Nesting is 0, and Max_Tasks is 0. The last three restrictions are checked prior to program execution.

Replace paragraph 3:

The following additional restrictions apply in this Annex.

by:

The following *restriction*_identifiers are language defined:

Delete paragraph 9:

No_Unchecked_Deallocation
Semantic dependence on Unchecked_Deallocation is not allowed.

Delete paragraph 16:

No_Unchecked_Conversion
Semantic dependence on the predefined generic Unchecked_Conversion is not allowed.

Replace paragraph 20:

No_IO
Semantic dependence on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, or Stream_IO is not allowed.

by:

No_IO
Semantic dependence on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, or Stream_IO is not allowed.

Insert before paragraph 24:

If an implementation supports pragma Restrictions for a particular argument, then except for the restrictions No_Unchecked_Deallocation, No_Unchecked_Conversion, No_Access_Subprograms, and No_Unchecked_Access, the associated restriction applies to the run-time system.

the new paragraph:

An implementation of this Annex shall support:

- the restrictions defined in this subclause; and

- the following restrictions defined in D.7: No_Task_Hierarchy, No_Abort_Statement, No_Implicit_Heap_Allocation; and
- the **pragma** Profile(Ravenscar); and
- the following uses of *restriction_parameter*_identifiers defined in D.7, which are checked prior to program execution:
  - Max_Task_Entries => 0,
  - Max_Asynchronous_Select_Nesting => 0, and
  - Max_Tasks => 0.

Insert after paragraph 27:

NOTES

10 Uses of *restriction_parameter*_identifier No_Dependence defined in 13.12.1: No_Dependence => Ada.Unchecked_Deallocation and No_Dependence => Ada.Unchecked_Conversion may be appropriate for high-integrity systems. Other uses of No_Dependence can also be appropriate for high-integrity systems.

H.5 Pragma Detect_Blocking

Insert new clause:

The following pragma forces an implementation to detect potentially blocking operations within a protected operation.

Syntax

The form of a pragma Detect_Blocking is as follows:

pragma Detect_Blocking;

Dynamic Semantics

An implementation is required to detect a potentially blocking operation within a protected operation, and to raise Program_Error (see 9.5.1).

Post-Compilation Rules

A pragma Detect_Blocking is a configuration pragma.

Implementation Permissions

An implementation is allowed to reject a compilation_unit if a potentially blocking operation is present directly within an entry_body or the body of a protected subprogram.

NOTES

10 An operation that causes a task to be blocked within a foreign language domain is not defined to be potentially blocking, and need not be detected.

H.6 Pragma Partition_Elaboration_Policy

Insert new clause:

This clause defines a pragma for user control over elaboration policy.

Syntax

The form of a pragma Partition_Elaboration_Policy is as follows:

pragma Partition_Elaboration_Policy (*policy*_identifier);

The *policy*_identifier shall be either Sequential, Concurrent or an implementation-defined identifier.

Post-Compilation Rules

A pragma Partition_Elaboration_Policy is a configuration pragma. It specifies the elaboration policy for a partition. At most one elaboration policy shall be specified for a partition.

If the Sequential policy is specified for a partition then pragma Restrictions (No_Task_Hierarchy) shall also be specified for the partition.

Dynamic Semantics

Notwithstanding what this International Standard says elsewhere, this pragma allows partition elaboration rules concerning task activation and interrupt attachment to be changed. If the *policy*_identifier is Concurrent, or if there is no pragma Partition_Elaboration_Policy defined for the partition, then the rules defined elsewhere in this Standard apply.

If the partition elaboration policy is Sequential, then task activation and interrupt attachment are performed in the following sequence of steps:

- The activation of all library-level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.
- The interrupt handlers are attached by the environment task.
- The environment task is suspended while the library-level tasks are activated.
- The environment task executes the main subprogram (if any) concurrently with these executing tasks.

If several dynamic interrupt handler attachments for the same interrupt are deferred, then the most recent call of Attach_Handler or Exchange_Handler determines which handler is attached.

If any deferred task activation fails, Tasking_Error is raised at the beginning of the sequence of statements of the body of the environment task prior to calling the main subprogram.

Implementation Advice

If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration then the partition is deadlocked and it is recommended that the partition be immediately terminated.

Implementation Permissions

If the partition elaboration policy is Sequential and any task activation fails then an implementation may immediately terminate the active partition to mitigate the hazard posed by continuing to execute with a subset of the tasks being active.

NOTES

7 If any deferred task activation fails, the environment task is unable to handle the Tasking_Error exception and completes immediately. By contrast, if the partition elaboration policy is Concurrent, then this exception could be handled within a library unit.

Annex J: Obsolescent Features

Replace paragraph 1:

This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this International Standard. Use of these features is not recommended in newly written programs.

by:

This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this International Standard. Use of these features is not recommended in newly written programs. Use of these features can be prevented by using pragma Restrictions(No_Obsolescent_Features), see 13.12.1.

J.9 The Storage_Size Attribute

Replace paragraph 3:

Storage_Size may be specified for a task first subtype via an attribute_definition_clause.

by:

Storage_Size may be specified for a task first subtype that is not an interface via an attribute_definition_clause.

J.10 Specific Suppression of Checks

Insert new clause:

Pragma Suppress can be used to suppress checks on specific entities.

Syntax

The form of a specific Suppress pragma is as follows:

pragma Suppress(identifier, [On =>] name);

Legality Rules

The identifier shall be the name of a check (see 11.5). The name shall statically denote some entity.

For a specific Suppress pragma that is immediately within a package_specification, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification.

Static Semantics

A specific Suppress pragma applies to the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification, to the end of the scope of the named entity. The pragma applies only to the named entity, or, for a subtype, on objects and values of its type. A specific Suppress pragma suppresses the named check for any entities to which it applies (see 11.5). Which checks are associated with a specific entity is not defined by this International Standard.

Implementation Permissions

An implementation is allowed to place restrictions on specific Suppress pragmas.

NOTES

3 An implementation may support a similar On parameter on pragma Unsuppress (see 11.5).

J.11 The Class Attribute of Untagged Incomplete Types

Insert new clause:

For the first subtype S of a type *T* declared by an incomplete_type_declaration that is not tagged, the following attribute is defined:

S'Class

Denotes the first subtype of the incomplete class-wide type rooted at *T*. The completion of *T* shall declare a tagged type. Such an attribute reference shall occur in the same library unit as the incomplete_type_declaration.

J.12 Pragma Interface

Insert new clause:

Syntax

In addition to an identifier, the reserved word **interface** is allowed as a pragma name, to provide compatibility with a prior edition of this International Standard.

J.13 Dependence Restriction Identifiers

Insert new clause:

The following restrictions involve dependence on specific language-defined units. The more general restriction No_Dependence (see 13.12.1) should be used for this purpose.

Static Semantics

The following *restriction*_identifiers exist:

No_Asynchronous_Control
: Semantic dependence on the predefined package Asynchronous_Task_Control is not allowed.

No_Unchecked_Conversion
: Semantic dependence on the predefined generic function Unchecked_Conversion is not allowed.

No_Unchecked_Deallocation
: Semantic dependence on the predefined generic procedure Unchecked_Deallocation is not allowed.

J.14 Character and Wide_Character Conversion Functions

Insert new clause:

The following declarations exist in the declaration of package Ada.Characters.Handling:

```
function Is_Character (Item : in Wide_Character) return Boolean
   renames Conversions.Is_Character;
function Is_String    (Item : in Wide_String)    return Boolean
   renames Conversions.Is_String;

function To_Character (Item       : in Wide_Character;
                       Substitute : in Character := ' ')  return Character
   renames Conversions.To_Character;

function To_String    (Item       : in Wide_String;
                        Substitute : in Character := ' ') return String
   renames Conversions.To_String;

function To_Wide_Character (Item : in Character) return Wide_Character
   renames Conversions.To_Wide_Character;

function To_Wide_String    (Item : in String)    return Wide_String
   renames Conversions.To_Wide_String;
```

Annex M: Implementation-Defined Characteristics

M Summary of Documentation Requirements

Replace paragraph 1:

The Ada language allows for certain machine dependences in a controlled manner. Each Ada implementation must document all implementation-defined characteristics.

by:

The Ada language allows for certain target machine dependences in a controlled manner. Each Ada implementation must document many characteristics and properties of the target system. This International Standard contains specific documentation requirements. In addition, many characteristics that require documentation are identified throughout this International Standard as being implementation defined. Finally, this International Standard requires documentation of whether implementation advice is followed. The following clauses provide summaries of these documentation requirements.

M.1 Specific Documentation Requirements

Insert new clause:

In addition to implementation-defined characteristics, each Ada implementation must document various properties of the implementation:

- [List of documentation requirements here]

M.2 Implementation-Defined Characteristics

Insert new clause:

The Ada language allows for certain machine dependences in a controlled manner. Each Ada implementation must document all implementation-defined characteristics:

- [List of characteristics here]

M.3 Implementation Advice

Insert new clause:

This International Standard sometimes gives advice about handling certain target machine dependences. Each Ada implementation must document whether that advice is followed:

- [List of advice here]

Annex N: Glossary

N Glossary

Replace paragraph 1:

This Annex contains informal descriptions of some terms used in this International Standard. To find more formal definitions, look the term up in the index.

by:

This Annex contains informal descriptions of some of the terms used in this International Standard. The index provides references to more formal definitions of all of the terms used in this International Standard.

Abstract type. An abstract type is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own.

Insert after paragraph 3:

Aliased. An aliased view of an object is one that can be designated by an access value. Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word **aliased**. The Access attribute can be used to create an access value designating an aliased object.

the new paragraph:

Ancestor. An ancestor of a type is the type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors. Note that ancestor and descendant are inverse relationships.

Insert after paragraph 4:

Array type. An array type is a composite type whose components are all of the same type. Components are selected by indexing.

the new paragraph:

Category (of types). A category of types is a set of types with one or more common properties, such as primitive operations. A category of types that is closed under derivation is also known as a *class*.

Replace paragraph 6:

Class. A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations.

by:

Class (of types). A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations.

Replace paragraph 8:

Composite type. A composite type has components.

by:

Composite type. A composite type may have components.

Delete paragraph 12:

Definition. All declarations contain a *definition* for a *view* of an entity. A view consists of an identification of the entity (the entity of the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see 8.5)).

Replace paragraph 13:

Derived type. A derived type is a type defined in terms of another type, which is the parent type of the derived type. Each class containing the parent type also contains the derived type. The derived type inherits properties such as components and primitive operations from the parent. A type together with the types derived from it (directly or indirectly) form a derivation class.

by:

Derived type. A derived type is a type defined in terms of one or more other types given in a derived type definition. The first of those types is the parent type of the derived type and any others are progenitor types. Each class containing the parent type or a progenitor type also contains the derived type. The derived type inherits properties such as components and primitive operations from the parent and progenitors. A type together with the types derived from it (directly or indirectly) form a derivation class.

Descendant. A type is a descendant of itself, its parent and progenitor types, and their ancestors. Note that descendant and ancestor are inverse relationships.

Replace paragraph 15:

Discriminant. A discriminant is a parameter of a composite type. It can control, for example, the bounds of a component of the type if that type is an array type. A discriminant of a task type can be used to pass data to a task of the type upon creation.

by:

Discriminant. A discriminant is a parameter for a composite type. It can control, for example, the bounds of a component of the type if the component is an array. A discriminant for a task type can be used to pass data to a task of the type upon creation.

Elaboration. The process by which a declaration achieves its run-time effect is called elaboration. Elaboration is one of the forms of execution.

Insert after paragraph 17:

Enumeration type. An enumeration type is defined by an enumeration of its values, which may be named by identifiers or character literals.

the new paragraph:

Evaluation. The process by which an expression achieves its run-time effect is called evaluation. Evaluation is one of the forms of execution.

Insert after paragraph 19:

Execution. The process by which a construct achieves its run-time effect is called *execution*. Execution of a declaration is also called *elaboration*. Execution of an expression is also called *evaluation*.

the new paragraph:

Function. A function is a form of subprogram that returns a result and can be called as part of an expression.

Insert after paragraph 20:

Generic unit. A generic unit is a template for a (nongeneric) program unit; the template can be parameterized by objects, types, subprograms, and packages. An instance of a generic unit is created by a generic_instantiation. The rules of the language are enforced when a generic unit is compiled, using a generic contract model; additional checks are performed upon instantiation to verify the contract is met. That is, the declaration of a generic unit represents a contract between the body of the generic and instances of the generic. Generic units can be used to perform the role that macros sometimes play in other languages.

the new paragraph:

Incomplete type. An incomplete type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Incomplete types can be used for defining recursive data structures.

Insert after paragraph 21:

Integer type. Integer types comprise the signed integer types and the modular types. A signed integer type has a base range that includes both positive and negative numbers, and has operations that may raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with "wraparound" semantics. Modular types subsume what are called "unsigned types" in some other languages.

the new paragraph:

Interface type. An interface type is a form of abstract tagged type which has no components or concrete operations except possibly null procedures. Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

Replace paragraph 23:

Limited type. A limited type is (a view of) a type for the assignment operation is not allowed. A nonlimited type is a (view of a) type for which the assignment operation is allowed.

by:

Limited type. A limited type is a type for which copying (such as in an assignment_statement) is not allowed. A nonlimited type is a type for which copying is allowed.

Insert after paragraph 24:

Object. An object is either a constant or a variable. An object contains a value. An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

the new paragraph:

Overriding operation. An overriding operation is one that replaces an inherited primitive operation. Operations may be marked explicitly as overriding or not overriding.

Insert after paragraph 25:

Package. Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

the new paragraph:

Parent. The parent of a derived type is the first type given in the definition of the derived type. The parent can be almost any kind of type, including an interface type.

Replace paragraph 29:

Private extension. A private extension is like a record extension, except that the components of the extension part are hidden from its clients.

by:

Private extension. A private extension is a type that extends another type, with the additional properties hidden from its clients.

Replace paragraph 30:

Private type. A private type is a partial view of a type whose full view is hidden from its clients.

by:

Private type. A private type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Private types can be used for defining abstractions that hide unnecessary details from their clients.

Procedure. A procedure is a form of subprogram that does not return a result and can only be called by a statement.

Progenitor. A progenitor of a derived type is one of the types given in the definition of the derived type other than the first. A progenitor is always an interface type. Interfaces, tasks, and protected types may also have progenitors.

Replace paragraph 33:

Protected type. A protected type is a composite type whose components are protected from concurrent access by multiple tasks.

by:

Protected type. A protected type is a composite type whose components are accessible only through one of its protected operations which synchronize concurrent access by multiple tasks.

Insert after paragraph 36:

Record type. A record type is a composite type consisting of zero or more named components, possibly of different types.

the new paragraph:

Renaming. A renaming_declaration is a declaration that does not define a new entity, but instead defines a view of an existing entity.

Insert after paragraph 37:

Scalar type. A scalar type is either a discrete type or a real type.

the new paragraph:

Subprogram. A subprogram is a section of a program that can be executed in various contexts. It is invoked by a subprogram call that may qualify the effect of the subprogram through the passing of parameters. There are two forms of subprograms: functions, which return values, and procedures, which do not.

Replace paragraph 38:

Subtype. A subtype is a type together with a constraint, which constrains the values of the subtype to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

by:

Subtype. A subtype is a type together with a constraint or null exclusion, which constrains the values of the subtype to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

Synchronized. A synchronized entity is one that will work safely with multiple tasks at one time. A synchronized interface can be an ancestor of a task or a protected type. Such a task or protected type is called a synchronized tagged type.

Replace paragraph 40:

Task type. A task type is a composite type whose values are tasks, which are active entities that may execute concurrently with other tasks. The top-level task of a partition is called the environment task.

by:

Task type. A task type is a composite type used to represent active entities which execute concurrently and which can communicate via queued task entries. The top-level task of a partition is called the environment task.

Replace paragraph 41:

Type. Each object has a type. A *type* has an associated set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. Types are grouped into *classes*. The types of a given class share a set of primitive operations. Classes are closed under derivation; that is, if a type is in a class, then all of its derivatives are in that class.

by:

Type. Each object has a type. A *type* has an associated set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. Types are grouped into *categories*. Most language-defined categories of types are also *classes* of types.

Replace paragraph 42:

View. (See **Definition**.)

by:

View. A view of an entity reveals some or all of the properties of the entity. A single entity may have multiple views.

Annex Q: Language-Defined Entities

Q Language-Defined Entities

Insert new clause:

This annex lists the language-defined entities of the language. A list of language-defined library units can be found in Annex A, "Predefined Language Environment".

Q.1 Language-Defined Packages

Insert new clause:

This clause lists all language-defined packages.

- [list of packages]

Q.2 Language-Defined Types and Subtypes

Insert new clause:

This clause lists all language-defined types and subtypes.

- [list of types]

Q.3 Language-Defined Subprograms

Insert new clause:

This clause lists all language-defined subprograms.

- [list of subprograms]

Q.4 Language-Defined Exceptions

Insert new clause:

This clause lists all language-defined exceptions.

- [list of exceptions]

Q.5 Language-Defined Objects

Insert new clause:

This clause lists all language-defined constants, variables, named numbers, and enumeration literals.

- [list of objects]