Information technology —Programming languages — Ada

AMENDMENT 1 (Draft 7)

Technologies de l'information —Langages de programmation — Ada

AMENDEMENT 1

Amendment 1 to International Standard ISO/IEC 8652:1995 was prepared by AXE Consultants.

© 2004, AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Compiled copies of standard library units and examples need not contain this copyright notice so long as the notice is included in all copies of the source code and documentation. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

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:

- The Ravenscar profile to provide a simplified tasking system for high-integrity systems (see clause D.13);
- A non-preemptive task dispatching policy (see clause D.2.4);
- 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);
- 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 context clauses that apply only in the private part of a package (see clause 10.1.2);
- Added standard packages, including time management (see 9.6), file directory and name management (see clause A.16), execution-time clocks (see clause D.14), timing events (see clause D.15), and array and vector operations (see clause G.3);
- Interfaces, to provide a limited form of multiple inheritance of operations (see clause 3.9.4); and
- A mechanism for writing C unions to make interfaces with C systems easier (see clause B.3.3).

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.`

Disclaimer:

This document is a draft of a possible amendment to Ada 95 (International Standard ISO/IEC 8652:1995). This draft contains only proposals substantially approved by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group (ARG). Many other important proposals are under

consideration by the ARG. Neither the ARG nor any other group has determined which, if any, of these proposals will be included in the amendment. Any proposal may be substantially changed or withdrawn before this document begins standardization, and other proposals may be added. This document is not an official publication or work product of the ARG.

Section 1: General

1.1.2 Structure

Replace paragraph 13: [AI95-00347-01]

- Annex H, ``Safety and Security"

by:

- Annex H, ``High Integrity Systems"

Section 2: Lexical Elements

2.9 Reserved Words

Replace paragraph 3: [AI95-00218-03; AI95-00251-01; AI95-00345-01]

NOTES

6 The reserved words appear in **lower case boldface** in this International Standard, except when used in the designator of an attribute (see 4.1.4). Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention — programs may be written in whatever typeface is desired and available.

by:

interface, **overriding**, and **synchronized** are nonreserved keywords.

NOTES

6 The reserved words appear in **lower case boldface** in this International Standard, except when used in the designator of an attribute (see 4.1.4). Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention — programs may be written in whatever typeface is desired and available.

Section 3: Declarations and Types

3.1 Declarations

Replace paragraph 3: [AI95-00348-01]

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

3.2 Types and Subtypes

Replace paragraph 4: [AI95-00326-01]

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, *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 provides its complete declaration. An incomplete or partial view is considered a composite type.

Replace paragraph 5: [AI95-00326-01]

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.

3.2.1 Type Declarations

Replace paragraph 4: [AI95-00251-01]

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 8: [AI95-00326-01]

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 as part of declaring an object of the type, is called a *full type*. A full type defines the *full view* of a 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: [AI95-00231-01]

subtype_indication ::= subtype_mark [constraint]

by:

subtype_indication ::=
[null_exclusion] subtype_mark [scalar_constraint | composite_constraint]

Delete paragraph 5: [AI95-00231-01]

constraint ::= scalar_constraint | composite_constraint

3.2.3 Classification of Operations

Insert after paragraph 6: [AI95-00335-01]

- For a specific type declared immediately within a package_specification, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same package_specification and that operate on the type;

the new paragraph:

- For a specific type, the stream-oriented attributes of the type that are available (see 13.13.2) at the end of the list of declarative_items where the type is declared;

Replace paragraph 7: [AI95-00200-01]

- 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:

- In the case of 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.

3.3.1 Object Declarations

Replace paragraph 5: [AI95-00287-01]

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 9: [AI95-00363-01]

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*.

3.4 Derived Types and Classes

Replace paragraph 2: [AI95-00251-01]

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

by:

interface_list ::= interface_subtype_mark {**and** interface_subtype_mark}

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

Replace paragraph 3: [AI95-00251-01]

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. A derived type has one parent type and zero or more interface ancestor types.

Replace paragraph 8: [AI95-00251-01]

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

by:

- Each class of types that includes the parent type or an interface ancestor type also includes the derived type.

Insert after paragraph 23: [AI95-00251-01]

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.

the new paragraph:

If a type declaration names an interface type in an interface_list, then the declared type inherits any user-defined primitive subprograms of the interface type in the same way.

Insert after paragraph 35: [AI95-00251-01]

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

the new paragraph:

18 An interface type which has an interface ancestor "is derived from" that type, and therefore is a derived type. A derived_type_definition, however, never defines an interface type.

3.4.1 Derivation Classes

Replace paragraph 2: [AI95-00251-01]

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 or interface type is also derived from each of its interface ancestor 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 6: [AI95-00230-01]

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 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).

Replace paragraph 10: [AI95-00251-01]

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 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 a descendant of any other type. Each untagged type has a unique ultimate ancestor.

3.5.4 Integer Types

Replace paragraph 16: [AI95-00340-01]

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.

3.5.9 Fixed Point Types

Replace paragraph 8: [AI95-00100-01]

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: [AI95-00230-01]

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

by:

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

Delete paragraph 11: [AI95-00363-01]

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: [AI95-00230-01]

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.

3.6.2 Operations of Array Types

Replace paragraph 16: [AI95-00287-01]

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.

3.7 Discriminants

Replace paragraph 1: [AI95-00326-01]

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 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: [AI95-00231-01]

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 9: [AI95-00231-01; AI95-00254-01]

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.

Delete paragraph 10: [AI95-00230-01]

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.

3.7.1 Discriminant Constraints

Replace paragraph 7: [AI95-00363-01]

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 a general access subtype, a discriminant_constraint is illegal if the designated type has defaults 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 defaults.

3.8 Record Types

Delete paragraph 8: [AI95-00287-01]

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

Replace paragraph 18: [AI95-00230-01]

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: [AI95-00287-01]

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 4: [AI95-00344-01]

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 any ancestors 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: [AI95-00362-01]

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

by:

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

Insert after paragraph 7: [AI95-00344-01]

```
    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;
```

Replace paragraph 12: [AI95-00344-01]

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.

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.

The function Is_Descendant_At_Same_Level returns True if 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.

3.9.1 Type Extensions

Replace paragraph 3: [AI95-00344-01; AI95-00345-01]

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 is nonlimited, then each of the components of the record_extension_part shall be nonlimited.

Replace paragraph 4: [AI95-00344-01]

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 of the generic unit.

3.9.2 Dispatching Operations of Tagged Types

Replace paragraph 17: [AI95-00196-01]

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: [AI95-00196-01; AI95-00239-01]

- 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 and is itself 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 and is the (possibly parenthesized or qualified) expression of an assignment statement whose target is of a class-wide type, then its controlling tag value is determined by the target;

3.9.3 Abstract Types and Subprograms

Replace paragraph 1: [AI95-00345-01]

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 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.

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: [AI95-00345-01]

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 4: [AI95-00251-01]

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 inherits a subprogram corresponding to an abstract subprogram or to a function with a controlling result, then

Replace paragraph 5: [AI95-00251-01]

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

by:

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

3.9.4 Interface Types

Insert new clause: [AI95-00251-01; AI95-00345-01]

An interface type is an abstract tagged type which provides a restricted form of multiple inheritance. A tagged, task, or protected type may be derived from one or more interface types.

Syntax

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

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 view of an object that is of a task interface type (or of a corresponding class-wide type) is a task object. Similarly, a view of an object that is of a protected interface type (or of a corresponding class-wide type) is a protected object.

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 derived from synchronized interfaces.

An interface type has no components.

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.

If a type declaration names an interface type in an interface_list, then the accessibility level of the declared type shall not be statically deeper than that of the interface type; also, the declared type shall not be declared in a generic body if the interface type is declared outside that body.

A descendant of a nonlimited interface shall be nonlimited. A descendant of a task interface shall be a task type or a task interface. A descendant of a protected interface shall be a protected type or a protected interface. A descendant of a synchronized interface shall be a task type, a protected type, or a synchronized interface.

A full view shall be a descendant of an interface type if and only if the corresponding partial view (if any) is also a descendant of the interface type.

For an interface type declared in a visible part, a primitive subprogram shall not be declared in the private part.

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.

3.10 Access Types

Replace paragraph 2: [AI95-00231-01]

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: [AI95-00231-01; AI95-00254-01]

access_definition ::= **access** subtype_mark

by:

null_exclusion ::= **not null**
access_definition ::=
  [null_exclusion] **access** [general_access_modifier] subtype_mark |
  [null_exclusion] **access** [**protected**] **procedure** parameter_profile |
  [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

Replace paragraph 9: [AI95-00225-01; AI95-00363-01]

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. A 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: [AI95-00230-01; AI95-00231-01; AI95-00254-01]

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 reserved word **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. If a null_exclusion is present, or the access_definition is for a controlling access parameter (see 3.9.2), the access_definition defines an access subtype which excludes the null value; otherwise the subtype includes a null value.

Replace paragraph 13: [AI95-00230-01; AI95-00231-01]

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. The null value of an 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 an access-to-object type, an allocator, which returns an access value designating a newly created object (see 3.10.2).

Replace paragraph 14: [AI95-00231-01]

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.

by:

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 first subtype of a type defined by an access_type_definition excludes the null value if a null_exclusion is present; otherwise, the first subtype includes the null value.

Legality Rules

A null_exclusion is only allowed in a subtype_indication whose subtype_mark denotes an access subtype that includes a null value.

Replace paragraph 15: [AI95-00231-01]

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 includes a null value. 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 satisifes a null_exclusion imposed on an access subtype if it does not equal the null value of its type.

Replace paragraph 17: [AI95-00230-01]

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 general access-to-variable type.

3.10.1 Incomplete Type Declarations

Replace paragraph 2: [AI95-00326-01]

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

by:

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

Replace paragraph 4: [AI95-00326-01]

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: [AI95-00326-01]

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:

Delete paragraph 7: [AI95-00326-01]

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

Replace paragraph 8: [AI95-00326-01]

- 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: [AI95-00326-01]

- 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 list of declarative_items 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 may not be deferred to the package body.

Replace paragraph 10: [AI95-00217-06; AI95-00326-01]

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

by:

A prefix shall not be of an incomplete view.

Replace paragraph 11: [AI95-00326-01]

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

by:

An incomplete_type_declaration declares an *incomplete view* of a type, and its first subtype; the first subtype is unconstrained if a known_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 in the immediate scope of the completion of T, or
- it occurs in 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.

3.10.2 Operations of Access Types

Replace paragraph 2: [AI95-00235-01]

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 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 12: [AI95-00230-01]

- 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 object (view).
- The accessibility level of the anonymous access type of an access discriminant specified for a limited type is the same as the containing object or associated constrained subtype. For other components having an anonymous access type, the accessibility level of the access type is the same as the level of the containing composite type.

Replace paragraph 13: [AI95-00254-01; AI95-00318-02]

- 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. If the actual is an allocator, this is the accessibility level of the execution of the called subprogram.
- The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is infinite.
- The accessibility level of the anonymous access type of an access result type (see 6.5) is the same as that of the associated function or access-to-subprogram type.

Replace paragraph 26: [AI95-00363-01]

- 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: [AI95-00363-01]

- 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:
  - *A*'s designated subtype 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 A's designated subtype shall be unconstrained.

Replace paragraph 32: [AI95-00229-01]

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, then the ultimate ancestor of *S* shall be a non-formal type declared within the generic unit.

Section 4: Names and Expressions

4.1.3 Selected Components

Insert after paragraph 9: [AI95-00252-01]

- 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 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 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.

Insert after paragraph 15: [AI95-00252-01]

For a selected_component that denotes a component of a variant, a check is made that the values of the discriminants are such that the value or object denoted by the prefix has this component. The exception Constraint_Error is raised if this check fails.

the new paragraph:

For a selected_component with a tagged prefix and selector_name that denotes a view of a subprogram, a call on the view denoted by the selected_component is equivalent to a call on the underlying subprogram with the first actual parameter being provided by the object or value denoted by the prefix (or the Access attribute of this object or value if the first formal is an access parameter), and the remaining actual parameters given by the actual_parameter_part, if any.

4.2 Literals

Delete paragraph 2: [AI95-00230-01]

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

Delete paragraph 7: [AI95-00230-01; AI95-00231-01]

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: [AI95-00230-01]

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: [AI95-00287-01]

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: [AI95-00287-01]

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

by:

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

Replace paragraph 8: [AI95-00287-01]

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: [AI95-00287-01]

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 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: [AI95-00287-01]

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: [AI95-00287-01]

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 a <>, 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).

4.3.2 Extension Aggregates

Replace paragraph 4: [AI95-00287-01]

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: [AI95-00306-01]

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: [AI95-00287-01]

```
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: [AI95-00287-01]

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

by:

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

Replace paragraph 7: [AI95-00287-01]

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.

Insert before paragraph 24: [AI95-00287-01]

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

the new paragraph:

Each array component expression defines the value for the associated component(s). For a component given by <>, the associated component(s) are initialized by default (see 3.3.1).

4.5.2 Relational Operators and Membership Tests

Replace paragraph 3: [AI95-00251-01]

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 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: [AI95-00230-01]

```
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: [AI95-00230-01]

```
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 the equality operators for *universal_access* shall be of a specific anonymous access type.

Legality Rules

The operands of the equality operators for *universal_access* shall be convertible to one another (see 4.6).

4.5.5 Multiplying Operators

Replace paragraph 20: [AI95-00364-01]

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. An explicit conversion is required on the

result when using the above fixed-fixed multiplication operator when either operand is of a type having a user-defined primitive multiplication operator declared immediately within the same list of declarations as the type and with both formal parameters of a fixed-point type. A corresponding requirement applies to the universal fixed-fixed division operator.

4.6 Type Conversions

Replace paragraph 8: [AI95-00251-01]

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.

If there is a type that is an ancestor of both the target type and the operand type, then

- The target type shall be untagged; or
- The operand type shall be covered by or descended from the target type; or
- 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.

If there is no type that is an ancestor of both the target type and the operand type, then

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

Replace paragraph 9: [AI95-00246-01; AI95-00251-01]

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

by:

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

Replace paragraph 10: [AI95-00251-01]

- The types shall have the same dimensionality;

by:

- The types shall have the same dimensionality;

Replace paragraph 11: [AI95-00251-01]

- Corresponding index types shall be convertible;

by:

- Corresponding index types shall be convertible;

Replace paragraph 12: [AI95-00246-01; AI95-00251-01]

- The component subtypes shall statically match; and

by:

- The component subtypes shall statically match;

Replace paragraph 12.1: [AI95-00246-01; AI95-00251-01; AI95-00363-01]

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

by:

- Neither the target type nor the operand type shall be limited; and
- In a view conversion: if the target type has aliased components, then so shall the operand type; and the operand type shall not have a tagged, private, or volatile subcomponent.

Replace paragraph 13: [AI95-00230-01; AI95-00251-01]

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

by:

- 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 not *universal_access*:

Delete paragraph 14: [AI95-00251-01; AI95-00363-01]

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

Replace paragraph 15: [AI95-00251-01]

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

by:

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

Replace paragraph 16: [AI95-00251-01; AI95-00363-01]

- 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

by:

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

Replace paragraph 17: [AI95-00251-01]

- 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.

by:

- 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.

Replace paragraph 18: [AI95-00230-01; AI95-00251-01]

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

by:

- 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 not *universal_access*:

Replace paragraph 19: [AI95-00251-01]

- The designated profiles shall be subtype-conformant.

by:

- The designated profiles shall be subtype-conformant.

Replace paragraph 20: [AI95-00251-01]

- 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.

by:

- 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.

Delete paragraph 21: [AI95-00251-01]

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:

Delete paragraph 22: [AI95-00251-01]

- The operand type shall be covered by or descended from the target type; or

Delete paragraph 23: [AI95-00251-01]

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

Delete paragraph 24: [AI95-00251-01]

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

Replace paragraph 49: [AI95-00230-01; AI95-00231-01]

- 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: [AI95-00231-01]

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 the null value, then a check is made that the value is not null.

4.8 Allocators

Replace paragraph 5: [AI95-00287-01; AI95-00344-01]

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.

Replace paragraph 6: [AI95-00363-01]

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: [AI95-00344-01]

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 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. 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 identified by the tag of the value of the qualified_expression is not deeper than that of the type of the allocator. Constraint_Error is raised if this check fails.

Replace paragraph 11: [AI95-00280-01]

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 created object contains any tasks, and the master of the type of the allocator has finished waiting for dependent tasks (see 9.3), 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.

If the object created by the 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.

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: [AI95-00263-01]

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.

Replace paragraph 38: [AI95-00268-01]

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 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, the rounding performed is implementation-defined. 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).

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 scalar 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 2: [AI95-00231-01; AI95-00254-01]

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

5.2 Assignment Statements

Replace paragraph 4: [AI95-00287-01]

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: [AI95-00287-01]

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.

Section 6: Subprograms

6.1 Subprogram Declarations

Replace paragraph 2: [AI95-00218-03]

subprogram_declaration ::= subprogram_specification ;

by:

overriding_indicator ::= [**not**] **overriding**
subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification ;

Replace paragraph 3: [AI95-00218-03]

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

by:

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

Replace paragraph 4: [AI95-00348-01]

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

by:

procedure_specification ::= **procedure** defining_program_unit_name parameter_profile

function_specification ::= **function** defining_designator parameter_and_result_profile

subprogram_specification ::=
    procedure_specification
  | function_specification

Replace paragraph 13: [AI95-00318-02]

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

by:

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

Replace paragraph 15: [AI95-00231-01]

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 23: [AI95-00231-01]

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.

Replace paragraph 24: [AI95-00231-01; AI95-00254-01]

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 parameter 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: [AI95-00254-01]

- 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.

6.3 Subprogram Bodies

Replace paragraph 2: [AI95-00218-03]

```
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: [AI95-00252-01]

- a subprogram declared immediately within a protected_body.

by:

- a subprogram declared immediately within a protected_body;
- the view of a subprogram denoted by a selected_component whose prefix denotes an object or value of a tagged type, and whose selector_name denotes a subprogram operating on the type (see 4.1.3).

Insert after paragraph 13: [AI95-00254-01]

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

the new paragraph:

- The calling convention for an access parameter of an access-to-subprogram type 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 16: [AI95-00318-02]

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, corresponding parameters have identical modes, and, for access parameters or access result types, the designated subtypes statically match.

Insert after paragraph 24: [AI95-00345-01]

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:

Two subprograms or entries are *type conformant* (respectively *mode conformant*, *subtype conformant*, or *fully conformant*) if their profiles are type conformant (respectively mode conformant, subtype conformant, or fully conformant).

6.4 Subprogram Calls

Replace paragraph 8: [AI95-00310-01]

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.

6.5 Return Statements

Replace paragraph 2: [AI95-00318-02]

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

by:

return_statement ::= simple_return_statement | extended_return_statement

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

extended_return_statement ::=
   return identifier : [**aliased**] *return*_subtype_indication [:= expression] [**do**
      handled_sequence_of_statements
   end return];

return_subtype_indication ::= subtype_indication | access_definition

Replace paragraph 3: [AI95-00318-02]

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 expression, if any, of a return_statement is called the *return expression*. The expected type for a return expression is the result type of the corresponding function.

Replace paragraph 4: [AI95-00318-02]

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:

If the result subtype of a function is limited at the point where the function is frozen (see 13.14), the result subtype shall be constrained.

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: [AI95-00318-02]

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 a return expression if and only if it applies to a function body. An extended_return_statement shall apply to a function body.

If the result subtype of a function is defined by a subtype_mark, the return_subtype_indication of an extended_return_statement that applies to the function body 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 a return 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.

If the type of the return expression is limited, then the return expression 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.

Static Semantics

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

Replace paragraph 6: [AI95-00318-02]

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 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) and becomes the initial value of the return object; otherwise, the return object is 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. The handled sequence of statements, if any, is then executed.

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

Delete paragraph 7: [AI95-00318-02]

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: [AI95-00318-02]

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.

Delete paragraph 9: [AI95-00318-02]

- 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: [AI95-00318-02]

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

Delete paragraph 11: [AI95-00318-02]

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

Delete paragraph 12: [AI95-00318-02]

- a tagged limited type;

Delete paragraph 13: [AI95-00318-02]

- a task or protected type;

Delete paragraph 14: [AI95-00318-02]

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

Delete paragraph 15: [AI95-00318-02]

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

Delete paragraph 16: [AI95-00318-02]

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

Delete paragraph 17: [AI95-00318-02]

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 19: [AI95-00318-02]

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

Replace paragraph 20: [AI95-00318-02; AI95-00344-01]

The exception Program_Error is raised if this check fails.

by:

If the result type is class-wide, a check is made that the accessibility level of the type identified by the tag of the result is not deeper than that of the master that elaborated the function body. If this check fails, Program_Error is raised.

Delete paragraph 21: [AI95-00318-02]

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: [AI95-00318-02]

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

by:

Finally, a transfer of control is performed which completes the execution of the construct to which the return_statement applies, and returns to the caller. In the case of a function, the function_call denotes a constant view of the return object.

Replace paragraph 24: [AI95-00318-02]

```
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_Mode;
end return;
```

6.5.1 Pragma No_Return

Insert new clause: [AI95-00329-01]

A pragma No_Return indicates that a procedure can return only by propagating an exception.

Syntax

The form of a pragma No_Return, which is a program unit pragma (see 10.1.5), is as follows:

pragma No_Return(*local*_name{, *local*_name});

Legality Rules

The pragma shall apply to one or more procedures or generic procedures.

If a pragma No_Return applies to a procedure or a generic procedure, there shall be no return_statements that apply to that procedure.

Static Semantics

If a pragma No_Return applies to a generic procedure, pragma No_Return applies to all instances of that generic procedure.

Dynamic Semantics

If a pragma No_Return applies to a procedure, then the exception Program_Error is raised at the point of the call of the procedure if the procedure body completes normally.

6.7 Null Procedures

Insert new clause: [AI95-00348-01]

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

Syntax

null_procedure_declaration ::= 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.

Section 7: Packages

7.3 Private Types and Private Extensions

Replace paragraph 2: [AI95-00251-01]

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**] **new** ancestor_subtype_indication [**and** interface_list] **with private**;

7.4 Deferred Constants

Replace paragraph 9: [AI95-00256-01]

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: [AI95-00287-01]

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.

Insert before paragraph 2: [AI95-00287-01; AI95-00318-02]

If a tagged record type has any limited components, then the reserved word **limited** shall appear in its record_type_definition.

the new paragraph:

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)

Insert after paragraph 8: [AI95-00287-01; AI95-00318-02]

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 by-reference type whose full view is limited 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: [AI95-00287-01; AI95-00318-02]

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: [AI95-00287-01]

- An initialization expression is not allowed in an object_declaration if the type of the object is limited.

Delete paragraph 11: [AI95-00287-01]

- A default expression is not allowed in a component_declaration if the type of the record component is limited.

Delete paragraph 12: [AI95-00287-01]

- An initialized allocator is not allowed if the designated type is limited.

Delete paragraph 13: [AI95-00287-01]

- A generic formal parameter of mode **in** must not be of a limited type.

Delete paragraph 14: [AI95-00287-01]

14 Aggregates are not available for a limited composite type. Concatenation is not available for a limited array type.

Delete paragraph 15: [AI95-00287-01]

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.

7.6 User-Defined Assignment and Finalization

Replace paragraph 5: [AI95-00161-01]

```
type Controlled is abstract tagged private;
```

by:

```
type Controlled is abstract tagged private;
pragma Preelaborable_Initialization(Controlled);
```

Replace paragraph 6: [AI95-00348-01]

```
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: [AI95-00161-01]

```
        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: [AI95-00348-01]

```
        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: [AI95-00348-01; AI95-00360-01]

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 21: [AI95-00147-01]

- 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 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 11: [AI95-00280-01]

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*.

Replace paragraph 16: [AI95-00256-01]

- 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.

Section 8: Visibility Rules

8.1 Declarative Region

Insert after paragraph 4: [AI95-00318-02]

- a loop_statement;

the new paragraph:

- an extended_return_statement;

8.3 Visibility

Insert after paragraph 12: [AI95-00251-01]

- 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 one is a non-null non-abstract subprogram, then it overrides all which are null or abstract subprograms.
  - If all are null procedures or abstract subprograms, then any null procedure overrides all abstract subprograms; if more than one homograph remains that is not thus overridden, then one is chosen arbitrarily to override the others.

Replace paragraph 20: [AI95-00217-06]

- 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 except at places that are within its declarative region or within the scope of a nonlimited_with_clause that mentions it. The limited view of a library package is hidden from all visibility except at places that are within the scope of a limited_with_clause that mentions it but not within the scope of a nonlimited_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.

Insert after paragraph 23: [AI95-00195-01]

- A declaration is also hidden from direct visibility where hidden from all visibility.

the new paragraph:

An attribute_definition_clause is *visible* at a place if a declaration at the point of the attribute_definition_clause would be immediately visible at the place.

Replace paragraph 26: [AI95-00218-01; AI95-00251-01]

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 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.

If two or more homographs are implicitly declared at the same place (and not overridden by a non-overridable declaration) then at most one shall be a non-null non-abstract subprogram. If all are null or abstract, then all of the null subprograms shall be fully conformant with one another. If all are abstract, then all of the subprograms shall be fully conformant with one another.

All of 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.

If a subprogram_declaration, abstract_subprogram_declaration, subprogram_body, subprogram_body_stub, subprogram_renaming_declaration, or generic_instantiation of a 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 point of the declaration or body;
- if the overriding_indicator is **not overriding**, then the operation shall not override any homograph (at any point).

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.

8.4 Use Clauses

Replace paragraph 5: [AI95-00217-06]

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: [AI95-00217-06]

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: [AI95-00217-06]

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: [AI95-00230-01]

```
object_renaming_declaration ::=
   defining_identifier : subtype_mark renames object_name;
```

by:

```
object_renaming_declaration ::=
   defining_identifier : subtype_mark renames object_name;
 | defining_identifier : access_definition renames object_name;
```

Replace paragraph 3: [AI95-00231-01; AI95-00254-01]

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 a specific anonymous access type which in the case of an access-to-object type shall have the same designated type as that of the access_definition and in the case of an access-to-subprogram type shall have a designated profile which is subtype conformant with that of the access_definition.

Replace paragraph 4: [AI95-00231-01; AI95-00254-01]

The renamed entity shall be an object.

by:

The renamed entity shall be an object. In the case where the type is defined by an access_definition of an access-to-object type, the renamed entity shall be of an access-to-constant type if and only if the access_definition defines an access-to-constant type.

Replace paragraph 5: [AI95-00363-01]

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: [AI95-00230-01]

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 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

Replace paragraph 3: [AI95-00217-06]

The renamed entity shall be a package.

by:

The renamed entity shall be a nonlimited view of a package.

8.5.4 Subprogram Renaming Declarations

Replace paragraph 2: [AI95-00218-03]

subprogram_renaming_declaration ::= subprogram_specification **renames** *callable_entity*_name;

by:

subprogram_renaming_declaration ::=
  [overriding_indicator]
  subprogram_specification **renames** *callable_entity*_name;

Insert after paragraph 5: [AI95-00228-01]

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:

If the *callable_entity*_name of a renaming denotes a subprogram which shall be overridden (see 3.9.3), then the renaming is illegal.

8.6 The Context of Overload Resolution

Replace paragraph 25: [AI95-00230-01; AI95-00231-01; AI95-00254-01]

- 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*, and that is access-to-constant only if *T* is access-to-constant; or
- when *T* is an anonymous access-to-subprogram type (see 3.10), to an access-to-subprogram type whose designated profile is subtype-conformant with that of *T*.

Section 9: Tasks and Synchronization

9.1 Task Units and Task Objects

Replace paragraph 2: [AI95-00345-01]

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];

Insert after paragraph 8: [AI95-00345-01]

A task declaration requires a completion, which shall be a task_body, and every task_body shall be the completion of some task declaration.

the new paragraphs:

Each interface_subtype_mark of an interface_list appearing within a task_type_declaration shall denote a limited interface type that is not a protected interface.

If a task_type_declaration includes an interface_list, then for each primitive subprogram inherited by the task type, at most one of the following shall apply:

- the inherited subprogram shall be 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 first parameter of the inherited subprogram shall be of the task type or an access parameter designating the task type, and there shall be an entry_declaration for a single entry with the same identifier and a profile that is type conformant with that of the inherited subprogram after omitting this first parameter, in which case the inherited subprogram is said to be *implemented* by the conforming entry, and its profile after omitting the first parameter shall be subtype conformant with that of the entry.

If neither applies, the inherited subprogram shall be a null procedure.

Insert after paragraph 9.1: [AI95-00345-01]

For a task declaration without a task_definition, a task_definition without task_items is assumed.

the new paragraph:

If a task_type_declaration includes an interface_list, the task type is derived from each interface named in the interface_list.

Replace paragraph 21: [AI95-00287-01]

> 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 has neither assignment 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).

9.4 Protected Units and Protected Objects

Replace paragraph 2: [AI95-00345-01]

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];

Insert after paragraph 10: [AI95-00345-01]

A protected declaration requires a completion, which shall be a protected_body, and every protected_body shall be the completion of some protected declaration.

the new paragraphs:

Each interface_subtype_mark of an interface_list appearing within a protected_type_declaration shall denote a limited interface type that is not a task interface.

If a protected_type_declaration includes an interface_list, then for each primitive subprogram inherited by the protected type, at most one of the following shall apply:

- the inherited subprogram shall be overridden with a primitive subprogram of the protected type; in this case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or
- the first parameter of the inherited subprogram shall be of the protected type or an access parameter designating the protected type, and there shall be a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected_type_declaration, having a profile that is type conformant with that of the inherited subprogram after omitting this first parameter; in this case the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry, and its profile after omitting the first parameter shall be subtype conformant with that of the protected subprogram or entry.

If neither applies, the inherited subprogram shall be a null procedure.

If an inherited subprogram is implemented by a protected procedure or entry, then its first parameter shall be an access-to-variable parameter, or of mode **out** or **in out**.

Replace paragraph 11: [AI95-00345-01]

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.

by:

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. If a protected_type_declaration includes an interface_list, the protected type is derived from each interface named in the interface_list.

Insert after paragraph 20: [AI95-00280-01]

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 23: [AI95-00287-01]

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 has neither assignment nor predefined equality operators.

9.6 Delay Statements, Duration, and Time

Replace paragraph 10: [AI95-00161-01]

```
package Ada.Calendar is
    type Time is private;
```

by:

```
package Ada.Calendar is
    type Time is private;
    pragma Preelaborable_Initialization(Time);
```

Replace paragraph 11: [AI95-00351-01]

```
  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: [AI95-00351-01]

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: [AI95-00351-01]

Static Semantics

The following language-defined library packages exist:

```
package Ada.Calendar.Time_Zones is

    -- Time zone manipulation:

    type Time_Offset is range -1440 .. 1440;

    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 -999 .. 999;

    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 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;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Second_Number;

function Sub_Second (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        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);

procedure Split (Date       : in Time;
                 Time_Zone  : in Time_Zones.Time_Offset := 0;
                 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);

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;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                         return Time;

procedure Split (Date       : in Time;
                 Time_Zone  : in Time_Zones.Time_Offset := 0;
                 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);

procedure Split (Date       : in Time;
```

```
                        Time_Zone  : in Time_Zones.Time_Offset := 0;
                        Year       : out Year_Number;
                        Month      : out Month_Number;
                        Day        : out Day_Number;
                        Seconds    : out Day_Duration;
                        Leap_Second: out Boolean);

    -- 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 Ada.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, 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 non-negative. 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 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;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Second_Number;
```

Returns the second within the hour and minute for Date, as appropriate for the specified time zone offset.

```
function Sub_Second (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Second_Duration;
```

Returns the fraction of second for Date (this has the same accuracy as Day_Duration), as appropriate for the specified time zone offset.

```
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 Hour:Minute:Second.Sub_Second. This value can be used in Calendar.Time_Of as well as the argument to Calendar."+" and Calendar."-".

```
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.Sub_Second.

```
procedure Split (Date       : in Time;
                 Time_Zone  : in Time_Zones.Time_Offset := 0;
                 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);
```

Splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset.

```
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;
```

Returns a Time built from the date and time values, relative to the specified time zone offset. Time_Error is raised if Leap_Second is True, and Hour, Minute, and Second are not appropriate for a Leap_Second.

```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration;
                  Leap_Second: Boolean := False;
```

```
                    Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Time;
```

Returns a Time built from the date and time values, relative to the specified time zone offset. Time_Error is raised if Leap_Second is True, and Seconds is not appropriate for a Leap_Second.

```
procedure Split (Date       : in Time;
                 Time_Zone  : in Time_Zones.Time_Offset := 0;
                 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);
```

Split Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset. Leap_Second is true if Date identifies a leap second.

```
procedure Split (Date       : in Time;
                 Time_Zone  : in Time_Zones.Time_Offset := 0;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Seconds    : out Day_Duration;
                 Leap_Second: out Boolean);
```

Split Date into its constituent parts (Year, Month, Day, Seconds), relative to the specified time zone offset. Leap_Second is true if Date identifies a leap second.

```
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 each value other than Year is a 2-digit form of the value of the functions defined in Calendar and Calendar.Formatting, including a leading '0', if needed. Year is a 4-digit value. If Include_Time_Fraction is True, Sub_Seconds*100 is suffixed to the string as a 2-digit value following a '.'.

```
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 each value is a 2-digit form of the value, including a leading '0', if needed. If Include_Time_Fraction is True, Sub_Seconds*100 is suffixed to the string as a 2-digit value following a '.'. If Elapsed_Time < 0.0, the result is Image (**abs** Elapsed_Time, Include_Time_Fraction) prefixed with "-". 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 time in the time zone known as Greenwich Mean Time (GMT) is generally equivalent to UTC time.

37 The implementation-defined time zone used for type Time may be, 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).

38 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: [AI95-00345-01]

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: [AI95-00345-01]

```
entry_call_alternative ::=
  entry_call_statement [sequence_of_statements]
```

by:

```
entry_call_alternative ::=
  procedure_or_entry_call_statement [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 denote an entry renamed as a procedure, a formal subprogram, 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: [AI95-00345-01]

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, and the delay_alternative sequence_of_statements is ignored.

9.7.4 Asynchronous Transfer of Control

Replace paragraph 4: [AI95-00345-01]

triggering_statement ::= entry_call_statement | delay_statement

by:

triggering_statement ::= procedure_or_entry_call_statement | delay_statement

Replace paragraph 6: [AI95-00345-01]

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 an procedure_or_entry_call_statement, 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). 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, and the abortable_part is never started.

9.8 Abort of a Task - Abort of a Sequence of Statements

Replace paragraph 3: [AI95-00345-01]

Each *task*_name is expected to be of any task type; they need not all be of the same task type.

by:

Each *task*_name is expected to be of any task type or task interface type; they need not all be of the same type.

9.9 Task and Entry Attributes

Replace paragraph 1: [AI95-00345-01]

For a prefix T that is of a task type (after any implicit dereference), the following attributes are defined:

by:

For a prefix T that is of a task type or task interface type (after any implicit dereference), the following attributes are defined:

Section 10: Program Structure and Compilation Issues

10.1.1 Compilation Units - Library Units

Insert after paragraph 12: [AI95-00217-06; AI95-00326-01]

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 is declared. 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 package is *not* the declaration of a library unit (the library package_declaration is); nonetheless, it is a library_item.

A library package_declaration is the completion of the declaration of its limited view.

The elaboration of the limited view of a package has no effect.

Replace paragraph 26: [AI95-00217-06]

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 implicit declaration of the limited view of a library package depends semantically upon the implicit declaration of the limited view of its parent. The declaration of a library package depends semantically upon the implicit 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.

10.1.2 Context Clauses - With Clauses

Replace paragraph 4: [AI95-00217-06; AI95-00326-01]

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: [AI95-00217-06]

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 is *named* in a with_clause if it is denoted by a *library_unit*_name in the with_clause. A library_item 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: [AI95-00217-06; AI95-00220-01; AI95-00262-01]

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, and the with_clause shall include the keyword **private**.

A name denoting a library item that is visible only due to being mentioned in with_clauses that include the keyword **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 a package_declaration[, not a subprogram_declaration, generic_declaration, generic_instantiation, or package_renaming_declaration].

A limited_with_clause shall not appear on a library_unit_body or subunit.

A limited_with_clause which names a library_item shall not appear:

- in the same context_clause as a nonlimited_with_clause which mentions the same library_item; or
- in the same context_clause as a use_clause which names an entity declared within the declarative region of the library_item; or
- in the scope of a nonlimited_with_clause which mentions the same library_item; or
- in the scope of a use_clause which names an entity declared within the declarative region of the library_item.

10.1.3 Subunits of Compilation Units

Replace paragraph 3: [AI95-00218-03]

subprogram_body_stub ::= subprogram_specification **is separate**;

by:

subprogram_body_stub ::=
[overriding_indicator]
subprogram_specification **is separate**;

Replace paragraph 8: [AI95-00243-01]

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. A *subunit of a program unit* includes subunits declared directly in the program unit as well as any subunits declared in those subunits (recursively).

10.1.4 The Compilation Process

Replace paragraph 3: [AI95-00217-06]

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 unit mentioned in a limited_with_clause within an environment are implementation defined.

Replace paragraph 6: [AI95-00217-06]

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: [AI95-00214-01]

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 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: [AI95-00212-01]

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 must still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options.

10.2 Program Execution

Replace paragraph 6: [AI95-00217-06]

- 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 limited view of a unit is needed, then the full view of the unit is needed.

Replace paragraph 9: [AI95-00256-01]

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: [AI95-00161-01]

A pragma Preelaborate is a library unit pragma.

the new paragraphs:

The form of pragma Preelaborable_Initialization is as follows:

pragma Preelaborable_Initialization (direct_name);

Replace paragraph 9: [AI95-00161-01]

- 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 which does not have preelaborable initialization. Similarly the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

Insert after paragraph 11: [AI95-00161-01]

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 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 or protected type) if the non-inherited components all have preelaborable initialization. Moreover, a user-defined controlled type with an overridding 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, or a record type whose components all have preelaborable initialization.

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 declarative_items of a package_specification, then the direct_name shall denote the first subtype of a private type, private extension, or protected type without entry_declarations, and the type shall be declared 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.

Section 11: Exceptions

11.3 Raise Statements

Replace paragraph 2: [AI95-00361-01]

raise_statement ::= **raise** [*exception*_name];

by:

raise_statement ::= **raise**; |
raise *exception*_name [**with** *string*_expression];

Insert after paragraph 3: [AI95-00361-01]

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: [AI95-00361-01]

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, a call of Ada.Exceptions.Exception_Message returns that string. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

11.4.1 The Package Exceptions

Replace paragraph 2: [AI95-00362-01]

```
package Ada.Exceptions is
    type Exception_Id is private;
    Null_Id : constant Exception_Id;
    function Exception_Name(Id : Exception_Id) return String;
```

by:

```
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;
```

Replace paragraph 3: [AI95-00362-01]

```
    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: [AI95-00329-01]

```
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 10: [AI95-00361-01; AI95-00378-01]

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. In this case, Exception_Message returns the Message parameter of Raise_Exception. For a raise_statement with an *exception*_name and a *string*_expression, Exception_Message returns that string. For a raise_statement with an *exception*_name but without a *string*_expression, Exception_Message returns 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: [AI95-00378-01]

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 Exception_Name functions return the full expanded name of the exception, in upper case, starting with a root library unit. The returned string has lower bound 1. 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.

Replace paragraph 13: [AI95-00378-01]

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: [AI95-00241-01; AI95-00329-01]

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.

11.4.2 Pragmas Assert and Assertion_Policy

Insert new clause: [AI95-00286-01]

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.

Legality Rules

The *policy*_identifier of an Assertion_Policy pragma 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(Ada.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 Ada.Assertions library unit.

The assertion policy that applies within an instance is the policy that applies within the generic unit.

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 it evaluates to False, evaluating the Message string, if any, and raising the exception Ada.Assertions.Assertion_Error, with a message if the Message argument is provided.

Calling the procedure Ada.Assertions.Assert without a Message parameter is equivalent to:

```
if Check = False then
```

```
      raise Ada.Assertions.Assertion_Error;
   end if;
```

Calling the procedure Ada.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 independent 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

Normally, the boolean expression in an Assert pragma 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.

11.5 Suppressing Checks

Replace paragraph 1: [AI95-00224-01]

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: [AI95-00224-01]

The form of a pragma Suppress is as follows:

by:

The forms of checking pragmas are as follows:

Replace paragraph 4: [AI95-00224-01]

pragma Suppress(identifier [, [On =>] name]);

by:

pragma Suppress(identifier);

pragma Unsuppress(identifier);

Replace paragraph 5: [AI95-00224-01]

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: [AI95-00224-01]

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: [AI95-00224-01]

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: [AI95-00224-01]

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 (see below), 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 27: [AI95-00224-01]

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: [AI95-00224-01]

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.

Section 12: Generic Units

12.3 Generic Instantiation

Replace paragraph 2: [AI95-00218-03]

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

Delete paragraph 8: [AI95-00287-01]

The type of a generic formal object of mode **in** shall be nonlimited.

Replace paragraph 9: [AI95-00255-01]

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 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; 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.

12.5 Formal Types

Replace paragraph 3: [AI95-00251-01]

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 8: [AI95-00233-01]

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 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 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 3: [AI95-00251-01]

formal_derived_type_definition ::= [**abstract**] **new** subtype_mark [**with private**]

by:

formal_derived_type_definition ::=
[**abstract**] **new** subtype_mark [[**and** interface_list] **with private**]

Insert after paragraph 10: [AI95-00231-01]

- 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.

Insert after paragraph 15: [AI95-00251-01]

For a generic formal type with an unknown_discriminant_part, the actual may, but need not, have discriminants, and may be definite or indefinite.

the new paragraph:

The actual type shall be a descendant of every ancestor of the formal type.

Replace paragraph 20: [AI95-00233-01]

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: [AI95-00233-01]

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 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 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.

12.5.4 Formal Access Types

Replace paragraph 4: [AI95-00231-01]

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: [AI95-00251-01; AI95-00345-01]

The class determined for a formal interface type is the class of all interface types.

Syntax

formal_interface_type_definition ::= interface_type_definition

Legality Rules

The actual type shall be an interface type.

The actual type shall be a descendant of every ancestor 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.

12.6 Formal Subprograms

Replace paragraph 3: [AI95-00348-01]

subprogram_default ::= default_name | <>

by:

subprogram_default ::= default_name | <> | **null**

Insert after paragraph 4: [AI95-00348-01]

default_name ::= name

the new paragraph:

A subprogram_default of **null** shall not be specified for a formal function.

Replace paragraph 9: [AI95-00345-01]

A formal_subprogram_declaration declares a generic formal subprogram. The types of the formal parameters and result, if any, of the formal subprogram are those determined by the subtype_marks given in the formal_subprogram_declaration; however, independent of the particular subtypes that are denoted by the subtype_marks, the nominal subtypes of the formal parameters and result, if any, are defined to be nonstatic, and unconstrained if of an array type (no applicable index constraint is provided in a call on a formal subprogram). In an instance, a formal_subprogram_declaration declares a view of the actual. The profile of this view takes its subtypes and calling convention from the original profile of the actual entity, while taking the formal parameter names and default_expressions from the profile given in the formal_subprogram_declaration. The view is a function or procedure, never an entry.

by:

A formal_subprogram_declaration declares a generic formal subprogram. The types of the formal parameters and result, if any, of the formal subprogram are those determined by the subtype_marks given in the formal_subprogram_declaration; however, independent of the particular subtypes that are denoted by the subtype_marks, the nominal subtypes of the formal parameters and result, if any, are defined to be nonstatic, and unconstrained if of an array type (no applicable index constraint is provided in a call on a formal subprogram). In an instance, a formal_subprogram_declaration declares a view of the actual. The profile of this view takes its subtypes and calling convention from the original profile of the actual entity, while taking the formal parameter names and default_expressions from the profile given in the formal_subprogram_declaration.

Insert after paragraph 10: [AI95-00348-01]

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 paragraph:

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.

Insert after paragraph 16: [AI95-00348-01]

18 The actual subprogram cannot be abstract (see 3.9.3).

the new paragraph:

19 A null procedure as a subprogram default has convention Intrinsic (see 6.3.1).

12.7 Formal Packages

Replace paragraph 3: [AI95-00317-01]

```
formal_package_actual_part ::=
   (<>) | [generic_actual_part]
```

by:

```
formal_package_actual_part ::=
   (<>)
  | [generic_actual_part]
  | ([generic_association {, generic_association},] others => <>)
```

Any positional generic_associations shall precede any named generic_associations.

Replace paragraph 5: [AI95-00317-01]

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 parameter of the formal package, determined as follows:

- If the formal_package_actual_part includes generic_associations as well as "**others** => <>", then only the actual parameters specified explicitly in these generic_associations are required to match;
- Otherwise, all actual parameters shall match, whether the 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 10: [AI95-00317-01]

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.

Section 13: Representation Issues

13.1 Representation Items

Replace paragraph 11: [AI95-00326-01]

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.

13.3 Representation Attributes

Delete paragraph 26: [AI95-00247-01]

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.

13.7 The Package System

Replace paragraph 3: [AI95-00362-01]

```
package System is
   pragma Preelaborate(System);
```

by:

```
package System is
   pragma Pure(System);
```

Replace paragraph 12: [AI95-00161-01]

```
    type Address is implementation-defined;
    Null_Address : constant Address;
```

by:

```
    type Address is implementation-defined;
    pragma Preelaborable_Initialization(Address);
    Null_Address : constant Address;
```

In paragraph 15 replace: [AI95-00221-01]

```
Default_Bit_Order : constant Bit_Order;
```

by:

```
Default_Bit_Order : constant Bit_Order := implementation-defined;
```

Replace paragraph 35: [AI95-00221-01]

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order.

by:

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order. Default_Bit_Order shall be a static constant.

Replace paragraph 36: [AI95-00362-01]

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 3: [AI95-00362-01]

```
package System.Storage_Elements is
   pragma Preelaborate(System.Storage_Elements);
```

by:

```
package System.Storage_Elements is
   pragma Pure(System.Storage_Elements);
```

Delete paragraph 15: [AI95-00362-01]

Package System.Storage_Elements may be declared pure.

13.9.1 Data Validity

Replace paragraph 12: [AI95-00167-01]

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.

13.11 Storage Management

Replace paragraph 6: [AI95-00161-01]

```
type Root_Storage_Pool is
    abstract new Ada.Controlled.Limited_Controlled with private;
```

by:

```
type Root_Storage_Pool is
    abstract new Ada.Controlled.Limited_Controlled with private;
pragma Preelaborable_Initialization(Root_Storage_Pool);
```

Replace paragraph 25: [AI95-00230-01]

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 initializing an access discriminant of an object of a limited type, and the discriminant is itself a subcomponent of an object being created by an outer allocator, then the storage pool used for the outer allocator should also be used for the allocator initializing the access discriminant;
- Otherwise, the storage pool should be created at the point of the allocator, and be reclaimed when the allocated object becomes inaccessible.

13.11.1 The Max_Size_In_Storage_Elements Attribute

Replace paragraph 3: [AI95-00256-01]

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. The value of this attribute is of type *universal_integer*.

13.12 Pragma Restrictions

Replace paragraph 4: [AI95-00381-01]

restriction ::= *restriction*_identifier
| *restriction_parameter*_identifier => expression

by:

restriction ::= *restriction*_identifier
| *restriction_parameter*_identifier => restriction_parameter_argument

```
restriction_parameter_argument ::= name | expression
```

Insert after paragraph 7: [AI95-00257-01; AI95-00368-01]

The set of restrictions is implementation defined.

the new paragraphs:

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.

13.12.1 Restriction No_Dependence

Insert new clause: [AI95-00381-01]

Static Semantics

The following *restriction_parameter*_identifier is language defined:

No_Dependence
: Specifies a language-defined library unit on which there are no semantic dependences.

Name Resolution Rules

The restriction_parameter_argument of a No_Dependence restriction shall be a name that corresponds to the full expanded name of a language-defined library unit.

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: [AI95-00161-01]

```
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: [AI95-00227-01]

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: [AI95-00227-01]

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: [AI95-00270-01]

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 operational 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, non-negative, and a multiple of Stream_Element'Size.

Implementation Advice

The recommended level of support for the Stream_Size attribute is: A Stream_Size clause should be supported for an elementary 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, floating point, ordinary fixed point, decimal fixed point, or access).

Replace paragraph 9: [AI95-00195-01; AI95-00270-01]

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, 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 the parent type of *T* is available anywhere within the immediate scope of *T*, and the attribute of the type of any of the extension components which are of a limited type, *L*, 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 first subtype low bound is non-negative, 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 17: [AI95-00270-01]

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.

by:

By default, the predefined stream-oriented attributes for an elementary type should only read or write 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.

Replace paragraph 27: [AI95-00195-01]

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, 7.6.1).

Insert after paragraph 28: [AI95-00260-01; AI95-00344-01]

For every subtype S'Class of a class-wide type *T*'Class:

the new paragraphs:

S'Class'Tag_Write

S'Class'Tag_Write denotes a procedure with the following specification:

```
procedure S'Class'Tag_Write (
    Stream : access Streams.Root_Stream_Type'Class;
    Tag : Ada.Tags.Tag);
```

S'Class'Tag_Write writes the value of Tag to Stream.

S'Class'Tag_Read

S'Class'Tag_Read denotes a function with the following specification:

```
function S'Class'Tag_Write (
     Stream : access Streams.Root_Stream_Type'Class;
     Ancestor : Ada.Tags.Tag)
     return Ada.Tags.Tag;
```

S'Class'Tag_Read reads a tag from Stream, and returns its value.

The default implementations of the Tag_Write and Tag_Read operate as follows:

- If *T* is a derived type with parent type *P*, the default implementation of Tag_Write calls *P*'Class'Tag_Write, and the default implementation of Tag_Read calls *P*'Class'Tag_Read;
- Otherwise, the default implementation of Tag_Write calls String'Output(*Stream*, Tags.External_Tag(*Tag*)) -- see 3.9. The default implementation of Tag_Read returns the value of Tags.Descendant_Tag(String'Input(*Stream*), Ancestor).

Replace paragraph 31: [AI95-00260-01; AI95-00344-01]

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:

Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. Otherwise, writes the external tag of *Item* to *Stream* (by calling S'Tag_Write(*Stream*, *Item*'Tag)) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag.

Replace paragraph 34: [AI95-00260-01; AI95-00344-01]

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 S'Tag_Read(*Stream*, S'Tag) which might raise Tag_Error) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; converts that result to S'Class and returns it.

Replace paragraph 35: [AI95-00195-01]

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: [AI95-00195-01]

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.

A stream-oriented attribute for a subtype of a specific type *T* is *available* at places where one of the following conditions is true:

- The attribute_designator is Read, Write or Output, and *T* is nonlimited.
- The attribute_designator is Input, and *T* is nonlimited and not abstract.
- 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.
- 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; or

- The attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible; or
- where 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.

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.

Insert after paragraph 36.1: [AI95-00195-01]

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).

Insert after paragraph 38: [AI95-00260-01]

User-specified attributes of S'Class are not inherited by other class-wide types descended from S.

the new paragraph:

User-specified Tag_Read and Tag_Write attributes should raise an exception if presented with a tag value not in S'Class.

13.14 Freezing Rules

Insert after paragraph 7: [AI95-00251-01]

- The declaration of a record extension causes freezing of the parent subtype.

the new paragraph:

- The declaration of a specific descendant of an interface type freezes the interface type.

Insert after paragraph 15: [AI95-00341-01]

- 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.

Annex A: Predefined Language Environment

A.3.2 The Package Characters.Handling

Replace paragraph 2: [AI95-00362-01]

```
package Ada.Characters.Handling is
   pragma Preelaborate(Handling);
```

by:

```
package Ada.Characters.Handling is
   pragma Pure(Handling);
```

A.4.2 The Package Strings.Maps

Replace paragraph 3: [AI95-00362-01]

```
package Ada.Strings.Maps is
   pragma Preelaborate(Maps);
```

by:

```
package Ada.Strings.Maps is
   pragma Pure(Maps);
```

Replace paragraph 4: [AI95-00161-01]

```
   -- 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: [AI95-00161-01]

```
   -- 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: [AI95-00301-01]

```
-- 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;

   function Index (Source  : in String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going   : in Direction := Forward)
      return Natural;

   function Index_Non_Blank (Source : in String;
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
```

Insert after paragraph 56: [AI95-00301-01]

- 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 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: [AI95-00301-01]

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). 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: [AI95-00301-01]

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: [AI95-00301-01]

```
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: [AI95-00301-01]

```
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;
    Drop   : in Truncation := Error)
       return Bounded_String;

procedure Bounded_Slice
   (Source : in     Bounded_String;
    Target :    out Bounded_String;
    Low    : in     Positive;
    High   : in     Natural;
    Drop   : in     Truncation := Error);
```

Replace paragraph 43: [AI95-00301-01]

```
-- 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;

   function Index (Source  : in Bounded_String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going   : in Direction := Forward)
      return Natural;

   function Index_Non_Blank (Source : in Bounded_String;
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
```

Insert after paragraph 92: [AI95-00301-01]

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: [AI95-00238-01; AI95-00301-01]

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;
    Drop   : in Truncation := Error)
      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;
    Drop   : in     Truncation := Error);
```

Equivalent to Target := Bounded_Slice (Soruce, Low, High, Drop);

A.4.5 Unbounded-Length String Handling

Replace paragraph 4: [AI95-00161-01]

```
type Unbounded_String is private;
```

by:

```
type Unbounded_String is private;
pragma Preelaborable_Initialization(Unbounded_String);
```

Insert after paragraph 11: [AI95-00301-01]

```
function To_String (Source : in Unbounded_String) return String;
```

the new paragraphs:

```
procedure Set_Unbounded_String
   (Target :    out Unbounded_String;
    Source : in     String;
    Drop   : in     Truncation := Error);
```

Insert after paragraph 22: [AI95-00301-01]

```
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;
     Drop   : in Truncation := Error)
        return Unbounded_String;

 procedure Unbounded_Slice
    (Source : in     Unbounded_String;
     Target :    out Unbounded_String;
     Low    : in     Positive;
     High   : in     Natural;
     Drop   : in     Truncation := Error);
```

Insert after paragraph 38: [AI95-00301-01]

```
-- 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;

   function Index (Source  : in Unbounded_String;
                   Set     : in Maps.Character_Set;
                   From    : in Positive;
                   Test    : in Membership := Inside;
                   Going    : in Direction := Forward)
      return Natural;

   function Index_Non_Blank (Source : in Unbounded_String;
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
```

Insert after paragraph 72: [AI95-00360-01]

```
   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: [AI95-00301-01]

- 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: [AI95-00301-01]

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: [AI95-00362-01]

```
   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 4: [AI95-00161-01]

```
      -- 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: [AI95-00161-01]

```
-- 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);
```

A.5.2 Random Number Generation

Insert after paragraph 15: [AI95-00360-01]

```
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: [AI95-00360-01]

```
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 Discrete_Random.

A.5.3 Attributes of Floating Point Types

Insert after paragraph 41: [AI95-00267-01]

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.8 Sequential and Direct Files

Replace paragraph 1: [AI95-00283-01]

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: [AI95-00360-01]

```
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: [AI95-00283-01]

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 22: [AI95-00248-01]

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: [AI95-00360-01]

```
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: [AI95-00301-01]

```
      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: [AI95-00360-01]

```
      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: [AI95-00223-01]

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: [AI95-00301-01]

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: [AI95-00301-01]

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.

The exception End_Error is propagated if an attempt is made to skip a file terminator.

A.10.11 Input-Output for Unbounded Strings

Insert new clause: [AI95-00301-01]

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 Text_IO.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 Text_IO.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

Insert after paragraph 3: [AI95-00301-01]

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.

the new paragraph:

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.

A.12.1 The Package Streams.Stream_IO

Insert after paragraph 27: [AI95-00360-01]

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: [AI95-00283-01]

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: [AI95-00085-01]

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: [AI95-00256-01]

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.16 The Package Directories

Insert new clause: [AI95-00248-01]

The package Ada.Directories provides operations for manipulating files and directories, and their names.

Static Semantics

The library package Ada.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);

   -- 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 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 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 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 if a portion of the contents of the directory are deleted.

```
procedure Delete_File (Name : in String);
```

Deletes an existing ordinary or special file with 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 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 New_Name.

```
procedure Copy_File (Source_Name, Target_Name : in String;
                     Form : in String);
```

Copies the contents of the existing external file with Source_Name to Target_Name. The resulting external file is a duplicate of the source external file. The Form 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 the creating of 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 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 discontiguous (not all elements exist), the result is implementation-defined. 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 the 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 (More_Entries returns False).

```
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 entry in the directory named by Directory for entries matching Pattern. Pattern represents a file name matching pattern. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation-defined. Only items which 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).

```
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.

```
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 is discontiguous (not all elements exist), the result is implementation-defined. 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 the 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 is available (such as the owner or creation date) in a directory entry, the implementation should provide functions in a child package Ada.Directories.Information to retrieve it.

Start_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.

For Rename, if both New_Name and Old_Name are simple names, then Rename should not propagate Use_Error.

NOTES

37 The file name 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.

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, Is_Directory will always return False, and Containing_Directory will always raise Use_Error.

40 If the target system does not support creation or deletion of directories, Create_Directory, Create_Path, Delete_Directory, and Delete_Tree will always propagate Use_Error.

Annex B: Interface to Other Languages

B.1 Interfacing Pragmas

Insert after paragraph 38: [AI95-00320-01]

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.

B.2 The Package Interfaces

Insert after paragraph 10: [AI95-00204-01]

- 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: [AI95-00204-01]

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: [AI95-00204-01]

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: [AI95-00376-01]

Interfacing with C

by:

Interfacing with C and C++

Replace paragraph 1: [AI95-00376-01]

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: [AI95-00376-01]

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.

Replace paragraph 50: [AI95-00258-01]

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.

Replace paragraph 60.2: [AI95-00216-01]

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.

B.3.1 The Package Interfaces.C.Strings

Replace paragraph 5: [AI95-00161-01]

```
type Chars_Ptr is private;
```

by:

```
type Chars_Ptr is private;
pragma Preelaborable_Initialization(Chars_Ptr);
```

Replace paragraph 6: [AI95-00276-01]

```
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: [AI95-00242-01]

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: [AI95-00216-01]

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 not be passed as a generic actual parameter if the corresponding formal type has a known_discriminant_part or is a formal derived type that is not an unchecked union type.

An unchecked union subtype shall only be passed as a generic actual parameter if the corresponding formal type does not have a known_discriminant_part, or is a formal derived type that is an unchecked union type.

Static Semantics

An unchecked union type is eligible for convention C.

Discriminant_Check is suppressed for an unchecked union type.

All objects of an unchecked union type have the same size.

Discriminants of objects of an unchecked union type are of size zero.

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.3.1 Protected Procedure Handlers

Replace paragraph 8: [AI95-00253-01]

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 a protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library level declaration. In addition, any object_declaration of such a type shall be a library level declaration.

C.4 Preelaboration Requirements

Insert after paragraph 4: [AI95-00161-01]

- 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.6 Shared Variable Control

Replace paragraph 7: [AI95-00272-01]

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: [AI95-00259-01]

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.7.1 The Package Task_Identification

Replace paragraph 2: [AI95-00362-01]

```
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;
```

C.7.3 Task Termination Procedures

Insert new clause: [AI95-00266-02]

This clause specifies a package for associating protected procedures with a task. One such procedure is invoked when the task is about to terminate.

Static Semantics

The following language-defined library package exists:

```
with System;
with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination is
  type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);

  type 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(New_Handler: in Handler);
  function Current_Task_Fallback_Handler return Handler;

  procedure Set_Specific_Handler(
      T           : in Ada.Task_Identification.Task_Id;
      New_Handler : in Handler);
  function Specific_Handler(T : Ada.Task_Identification.Task_Id)
      return Handler;
end Ada.Task_Termination;
```

Dynamic Semantics

A call of Set_Dependents_Fallback_Handler sets the fall-back handler for all dependent tasks. If a fall-back handler had previously been set it is replaced. A call with a null access parameter is equivalent to removing the fall-back handler. A call of Current_Task_Fallback_Handler returns the fall-back handler that is currently in effect for the calling task. If no fall-back handler has been set it returns null.

A call of Set_Specific_Handler sets a specific handler for the task identified by T. If a specific handler had previously been set it is replaced. A call with a null access parameter is equivalent to removing the specific handler. A call of Specific_Handler returns the specific handler if one has been set, otherwise the handler returned is null.

As part of the finalization of a task_body, after performing the actions specified in 7.6 for finalization of a master, the task specific handler, if not null, is called. If there is no such specific handler, a fall-back handler

is determined by recursively searching for a non null fall-back handler in the tasks upon which it depends. If such a fall-back handler is determined it is executed; otherwise no handler is executed.

If the task completed due to completing the last statement of the task 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 completion is due to abort then Cause is set to Abnormal and X is set to Null_Occurrence. If completion is due to an unhandled exception then Cause is set to Unhandled_Exception and the associated exception occurrence is passed.

For all the operations defined in this package, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Null_Task_ID.

An exception propagated from a handler that is invoked as part of a task's termination has no effect.

Annex D: Real-Time Systems

D.2 Priority Scheduling

Replace paragraph 1: [AI95-00321-01]

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: [AI95-00321-01]

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.

Replace paragraph 2: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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 Pragma Task_Dispatching_Policy

Replace the title: [AI95-00321-01]

The Standard Task Dispatching Policy

by:

Pragma Task_Dispatching_Policy

Replace paragraph 3: [AI95-00321-01]

The *policy*_identifier shall either be FIFO_Within_Priorities or an implementation-defined identifier.

by:

The *policy*_identifier shall either be one defined in this Annex or an implementation-defined identifier.

Delete paragraph 5: [AI95-00321-01]

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.

Delete paragraph 7: [AI95-00321-01]

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: [AI95-00321-01]

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

Delete paragraph 9: [AI95-00321-01]

- 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: [AI95-00321-01]

- 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: [AI95-00321-01]

- 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: [AI95-00321-01]

Each of the events specified above is a task dispatching point (see D.2.1).

Delete paragraph 13: [AI95-00321-01]

In addition, when a task is preempted, it is added at the head of the ready queue for its active priority.

Delete paragraph 14: [AI95-00321-01]

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: [AI95-00321-01]

- 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: [AI95-00321-01]

- 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: [AI95-00256-01; AI95-00321-01]

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.

Delete paragraph 18: [AI95-00321-01]

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.

Delete paragraph 19: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

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: [AI95-00321-01]

This clause defines the policy_identifier, FIFO_Within_Priorities.

Post-Compilation Rules

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.

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.

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: [AI95-00298-01]

A non-preemptive dispatching policy is defined by *policy*_identifier Non_Preemptive_FIFO_Within_Priorities.

Legality Rules

Non_Preemptive_FIFO_Within_Priorities can be specified as the *policy*_identifier of pragma Task_Dispatching_Policy (see D.2.2).

Post-Compilation Rules

If Non_Preemptive_FIFO_Within_Priorities is specified for a partition then Ceiling_Locking (see D.3) shall also be specified for that partition.

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. This is a task dispatching point (see D.2.1).

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 protected object does not contain pragma Interrupt_Priority, Interrupt_Handler or Attach_Handler.

D.3 Priority Ceiling Locking

Insert after paragraph 13: [AI95-00327-01]

- 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: [AI95-00256-01]

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 15: [AI95-00256-01]

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.

D.5 Dynamic Priorities

Insert before paragraph 1: [AI95-00327-01]

This clause specifies how the base priority of a task can be modified or queried at run time.

the new paragraphs:

This clause describes how the priority of an entity can be queried or modified at run time.

D.5.1 Dynamic Priorities for Tasks

[This changes the subclause of all of the existing text.]

Replace paragraph 3: [AI95-00362-01]

```
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);
```

Delete paragraph 11: [AI95-00327-01]

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.

D.5.2 Dynamic Priorities for Protected Objects

Insert new clause: [AI95-00327-01]

This clause specifies how the priority of a protected object can be modified or queried at run time.

Static Semantics

The following attribute of a protected object is defined:

P'Priority

Denotes a non-aliased component of the enclosing protected object P. This component is of type System.Any_Priority and its value is the priority of P. Reference to this attribute shall appear only inside the body of P.>

The initial value of this attribute is set by pragmas Priority or Interrupt_Priority, and can be changed by an assignment.

Dynamic Semantics

If the Locking_Policy Ceiling_Locking 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.

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);
private
   null;
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 The value of P'Priority 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: [AI95-00360-01]

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 (see 7.6) shall be declared only at library level.

Replace paragraph 9: [AI95-00327-01]

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 attribute 'Priority.

Insert after paragraph 10: [AI95-00305-01; AI95-00353-01]

No_Asynchronous_Control
: There are no semantic dependences on the package Asynchronous_Task_Control.

the new paragraphs:

No_Calendar
: There are no semantic dependencies on package Ada.Calendar.

No_Dynamic_Attachment
: There is no call to any of the operations defined in package Ada.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_Protected_Type_Allocators
: There are no allocators for protected types or types containing protected type components.

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_Synchronous_Control
: There are no semantic dependences on the package Synchronous_Task_Control.

No_Task_Attributes_Package
: There are no semantic dependencies on package Ada.Task_Attributes.

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: [AI95-00305-01]

This paragraph was deleted

by:

No_Task_Termination
: All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate.

Insert after paragraph 19: [AI95-00305-01]

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.

D.10 Synchronous Task Control

Replace paragraph 3: [AI95-00362-01]

```
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: [AI95-00362-01]

```
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;
```

D.13 Run-time Profiles and the Ravenscar Profile

Insert new clause: [AI95-00249-01; AI95-00297-01]

This clause specifies a mechanism for defining run-time profiles. It also defines one such profile, Ravenscar.

Syntax

The form of a pragma Profile is as follows:

pragma Profile (*profile*_identifier [profile_argument_associations]);

profile_argument_associations ::= pragma_argument_association, {pragma_argument_association}

Legality Rules

The *profile*_identifier shall be either Ravenscar or an implementation-defined identifier. For *profile*_identifier Ravenscar, there shall be no profile_argument_associations. For other *profile*_identifiers, the semantics of any profile_argument_associations are implementation-defined.

Static Semantics

A profile is equivalent to the set of configuration pragmas that is defined for each *profile*_identifier. The *profile*_identifier 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 (
                  Max_Entry_Queue_Length => 1,
                  Max_Protected_Entries => 1,
                  Max_Task_Entries => 0,
                  No_Abort_Statements,
                  No_Asynchronous_Control,
                  No_Calendar,
                  No_Dynamic_Attachment,
                  No_Dynamic_Priorities,
                  No_Implicit_Heap_Allocations,
                  No_Local_Timing_Events,
                  No_Local_Protected_Objects,
                  No_Protected_Type_Allocators,
                  No_Relative_Delay,
                  No_Requeue_Statements,
                  No_Select_Statements,
                  No_Task_Allocators,
                  No_Task_Attributes_Package,
                  No_Task_Hierarchy,
                  No_Task_Termination,
                  Simple_Barriers);
```

Post-Compilation Rules

A pragma Profile is a configuration pragma. There may be more than one pragma Profile for a partition.

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: [AI95-00307-01]

This clause specifies an execution-time clock package.

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 : CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

   function Time_Of (SC : Seconds_Count; TS : Time_Span) return CPU_Time;

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

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 behalf of it. 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 the type CPU_Time 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 some unspecified point between the task creation and the start of the task's activation.

CPU_Time_First and CPU_Time_Last are the smallest and largest values of the 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 amount of execution time of the task associated with the execution-time clock of the task represented by T.

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.

For all the operations and types defined in this package, 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.

Erroneous Execution

If a value of Task_ID is passed as a parameter to any of the subprograms of this package or used to define an object declared by a type provided by this package (or any language-defined child package of this package) and the corresponding task object 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: [AI95-00307-01]

This clause specifies a child of Execution_Time that provides a facility for calling a handler when a task has used a defined quantity of CPU time.

Static Semantics

The following language-defined library package exists:

```
with Ada.Task_Identification;
with System;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time.Timers is

   type Timer (T : access Ada.Task_Identification.Task_ID) is
      limited private;

   type Handler is not null access protected procedure (TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority :=
    implementation-defined;

   procedure Arm (TM: in out Timer; Interval : Time_Span; H : Handler);
   procedure Arm (TM: in out Timer; Abs_Time : CPU_Time;  H : Handler);
   procedure Disarm(TM : in out Timer);

   function Timer_Has_Expired(TM : Timer) return Boolean;
   function Time_Remaining(TM : Timer) return Time_Span;

   Timer_Error : exception;
   Timer_Resource_Error : exception;

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

The type Timer needs finalization (see 7.6).

Dynamic Semantics

Type Timer represents a software object that is capable of detecting execution time overruns. Each timer is attached to a specific execution time clock, which is the clock of the task specified by the access discriminant T. This type has operations for the application tasks to arm or disarm a timer, and to determine whether a timer has expired or not (Timer_Has_Expired). In addition, when arming the timer, an access to a protected procedure (type Handler) is passed. This procedure will be called by the implementation when the timer expires.

The constant Min_Handler_Ceiling is the priority value that will insure that no ceiling violation will occur when a handler is executed.

When a Timer object is created, or upon the first call to one of its Arm procedures, the resources required to operate a CPU-time timer based on the associated execution-time clock will be allocated and initialized. The timer is initialized in the disarmed state. If this operation would exceed the limit of the maximum number of timers in the system, the Timer_Resource_Error exception is raised.

The Arm procedure that takes a Time_Span parameter loads the associated timer with the relative value specified by Interval and sets it to the armed state. In this state the timer counts execution time and, when the CPU clock associated with the timer measures the passage of Interval, it is said to have expired. If the timer was already armed, it is rearmed.

The Arm procedure that takes a CPU_Time parameter loads the associated timer with the absolute value specified by Abs_Time and sets it to the armed state. In this state the timer monitors execution time and, when the CPU clock associated with the timer reaches the value Abs_Time, it is said to have expired. If the

value of Abs_Time had already been reached by the clock at the time of the call, the timer is set to the armed state and is said to have expired. If the timer was already armed, it is rearmed.

When a Timer expires, the protected procedure Handler is invoked by the system with a parameter equal to the Timer.

The Disarm procedure sets the timer to the disarmed state. In this state the timer will not expire.

The Time_Has_Expired function returns True if the timer is in the armed state and has expired, and returns False if the timer is in the armed state but has not yet expired. If the timer is in the disarmed state, the Timer_Error exception is raised.

The Time_Remaining function returns, when the timer is in the armed state, the CPU time interval that remains until the timer will expire, or a value representing zero if the timer has expired. If the timer is in the disarmed state, the Timer_Error exception is raised.

The Timer_Error exception is raised by Timer_Has_Expired, or Time_Remaining if an attempt is made to use a timer that is in the disarmed state.

For all the operations and types defined in this package, 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.

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.

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 the Timer_Resource_Error exception is raised.

D.14.2 Group Execution Time Budgets

Insert new clause: [AI95-00354-01]

This clause specifies a group execution time control package.

Static Semantics

The following language-defined library package exists:

```
with System;
with Ada.Task_Identification;
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is limited private;

  type Handler is not null 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 : Ada.Task_Identification.Task_ID);
  procedure Remove_Task(GB: in out Group_Budget;
                     T : 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 : Time_Span);
   procedure Add(GB: in out Group_Budget; Interval : 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; H : Handler);
   function Current_Handler(GB: Group_Budget) return 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 a CPU budget to be used by a group of tasks. This type needs finalization (see 7.6).

An object of type Group_Budget is said to be set if it has a registered Handler. An object is said to be cleared if it has no Handler. All Group_Budget objects are initially cleared.

Dynamic Semantics

Tasks of any priority are added to a group by calling Add_Task. Tasks are members of at most one group. Group_Budget_Error is raised by a call to Add_Task if the task is already a member of any group.

Tasks are removed from a group by calling Remove_Task. An attempt to remove a task that has not been added to the group will cause Group_Budget_Error to be raised.

The Is_Member function will return True if the task parameter is a member of the specified group. The Is_A_Group_Member function returns True if the task is a member of any group. Both return False otherwise.

The Members function returns the task IDs of the members of the group.

When a call to Replenish is made, the Group_Budget is loaded with the Time_Span value passed as a parameter. Any execution of the group of tasks results in the Group_Budget counting down. When the budget is exhausted (goes to Time_Span_Zero) the handler, if set, is called; the tasks continue to execute. A Group_Budget is initially loaded with zero budget.

A call to Budget_Remaining returns the remaining budget. If the budget is exhausted it will return Time_Span_Zero. This is the minimum value for the budget. A call to Budget_Has_Expired will return True if the budget is exhausted (equal to Time_Span_Zero), otherwise it returns False.

A Group_Budget can have its budget increased by calling Add. A negative value for the parameter will reduce the budget, but never below Time_Span_Zero.

A call of Replenish with a non positive value of To will causes exception Group_Budget_Error to be raised. A call to Add that results in the value of the budget going to Time_Span_Zero will cause the handler, if set, to be executed.

A call to Set_Handler registers the Handler and returns when GB is set. A call to Set_Handler for a Group_Budget that is already set, initially clears the Group_Budget then registers the new Handler.

A call to Current_Handler returns with the current Handler. If the Group_Budget denoted by GB is not set, exception Group_Budget_Error is raised.

A call to Cancel_Handler returns after the Group_Budget denoted by GB is cleared. Cancelled is assigned True if GB was set prior to it being cleared; otherwise the parameter is assigned False.

The constant Min_Handler_Ceiling is the priority value that will insure that no ceiling violation will occur when a handler is executed.

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 the 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 and types defined in this package, 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.

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.

D.15 Timing Events

Insert new clause: [AI95-00297-01]

This clause introduces a language-defined child package of Ada.Real_Time to allow user-defined protected procedures to be executed at a specified time without the need to use a task or a delay statement.

Static Semantics

The following language-defined package exists:

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is 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 Is_Handler_Set(Event : Timing_Event) return Boolean;
  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 needs finalization (see 7.6).

An object of type Timing_Event is said to be *set* if it has a registered Timing_Event_Handler. An object is said to be *cleared* if it has no registered Timing_Event_Handler. All Timing_Event objects are initially cleared.

Dynamic Semantics

A call to a Set_Handler procedure returns after the Timing_Event_Handler denoted by Handler is registered. The first Set_Handler procedure registers the Timing_Event_Handler for execution at time At_Time. The second Set_Handler procedure registers the Timing_Event_Handler for execution at time Ada.Real_Time.Clock + In_Time;

A call of either Set_Handler procedure is not a potentially blocking operation.

As soon as possible after the time registered for the event, the Timing_Event_Handler procedure is executed. The Timing_Event_Handler is only executed if the timing event is set at the time of execution. As the initial action of the execution of the Timing_Event_Handler the Timing_Event denoted by Event is cleared.

If the Ceiling_Locking policy (see D.3) is in effect when a Set_Handler procedure is called, a check is made that the ceiling priority of Timing_Event_Handler is Interrupt_Priority'last. If the check fails, Program_Error is raised.

If a Set_Handler procedure is called with zero or negative In_Time or with At_Time indicating a time in the past then Timing_Event_Handler is executed immediately by the task executing the Set_Handler call. The Timing_Event denoted by Event is cleared and the handler is not registered.

An exception propagated from a Timing_Event_Handler invoked by a timing event has no effect.

A call to Is_Handler_Set returns True if Event is set; otherwise it returns False.

A call to Current_Handler returns with the current Timing_Event_Handler. If the Timing_Event denoted by Event is not set, Current_Handler returns null.

A call to Cancel_Handler returns after the Timing_Event denoted by Event is cleared. Cancelled is assigned True if Event was set prior to it being cleared; otherwise the parameter is assigned False.

A call to Time_Of_Event returns with the time of Event. If Event is not set, Time_Of_Event returns Ada.Real_Time.Time_First.

As the final step of finalization of an object of type Timing_Event, the Timing_Event is cleared.

If several timing events are registered for the same time, they are executed in FIFO order of registration.

Metrics

The Implementation shall document the following metric:

- An upper bound on the lateness of the execution of a registered handler. That is, the maximum time between when a handler is actually executed and the time specified in the registration of that handler.

Implementation Advice

The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.

NOTES

Since a call to Set_Handling is a not a blocking operation, it can be called from within an object of type Timing_Event_Handler.

Annex E: Distributed Systems

E.2.2 Remote Types Library Units

Replace paragraph 8: [AI95-00240-01]

- 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:

- 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 Read and Write attributes specified by a visible attribute_definition_clause.

Replace paragraph 14: [AI95-00240-01]

- 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 have either a nonlimited type or a type with available Read and Write attributes (see 13.13.2);

E.2.3 Remote Call Interface Library Units

Replace paragraph 14: [AI95-00240-01]

- 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 formal parameter of a limited type unless that limited type has available Read and Write attributes (see 13.13.2);

E.5 Partition Communication Subsystem

Replace paragraph 1: [AI95-00273-01]

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: [AI95-00273-01]

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

No changes in this section.

Annex G: Numerics

G.1.1 Complex Types

Replace paragraph 4: [AI95-00161-01]

```
type Imaginary is private;
```

by:

```
type Imaginary is private;
pragma Preelaborable_Initialization(Imaginary);
```

G.1.2 Complex Elementary Functions

Replace paragraph 15: [AI95-00185-01]

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: [AI95-00185-01]

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: [AI95-00185-01]

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: [AI95-00185-01]

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 Ada.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.)

G.1.3 Complex Input-Output

Insert before paragraph 10: [AI95-00328-01]

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.

G.2.2 Model-Oriented Attributes of Floating Point Types

Replace paragraph 3: [AI95-00256-01]

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 Machine_Radix is a positive power of 10 and 1 otherwise. 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*.

G.3 Vector and Matrix Manipulation

Insert new clause: [AI95-00296-01]

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: [AI95-00296-01]

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;

   -- 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 Ada.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 functions is as described below. In most cases the functions 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 "*" (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 matrix 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 matrix 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 matrix 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 X'Range. 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 those of X. 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 those of A. 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'Machine_Mantissa)

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 may be performed by using the equality operator to compare the relevant components.

G.3.2 Complex Vectors and Matrices

Insert new clause: [AI95-00296-01]

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;

   -- 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 Ada.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 "+" (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 matrix 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. The exception 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 matrix 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 X'Range. 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 those of X. 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 those of A. 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'Machine_Mantissa) for mixed complex and real operands

g = sqrt(2.0) * X'Length * Real'Machine_Radix**(1-Real'Machine_Mantissa) 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 may 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: [AI95-00347-01]

Safety and Security

by:

High Integrity Systems

Replace paragraph 1: [AI95-00347-01]

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.3.1 Pragma Reviewable

Replace paragraph 8: [AI95-00209-01]

- 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 9: [AI95-00209-01]

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: [AI95-00347-01]

Safety and Security Restrictions

by:

High Integrity Restrictions

Replace paragraph 2: [AI95-00347-01]

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.

by:

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. Pragma Profile(Ravenscar) applies in this Annex.

H.5 Pragma Detect_Blocking

Insert new clause: [AI95-00305-01]

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: [AI95-00265-01]

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

The pragma is a configuration pragma. It applies to all compilation units in 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, all task activations for library-level tasks and all interrupt handler attachments for library-level interrupt handlers are deferred. The deferred task activations and handler attachments occur after the elaboration of all library_items prior to calling the main subprogram. At this point the Environment task is suspended until all deferred task activations and handler attachments are complete.

If any deferred task activation fails, Tasking_Error is raised in the Environment task. The Environment task and all tasks whose activations fail are terminated. If a number of 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.

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 Permission

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.

Annex J: Obsolescent Features

Replace paragraph 1: [AI95-00368-01]

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.

J.10 Specific Suppression of Checks

Insert new clause: [AI95-00224-01]

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: [AI95-00326-01]

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.