

# Programming languages — Ada

## TECHNICAL CORRIGENDUM 1 (Draft 1)

*Langages de programmation — Ada*

*RECTIFICATIF TECHNIQUE 1*

Technical Corrigendum 1 to International Standard ISO/IEC 8652:2012 was prepared by AXE Consultants.

© 2015, 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. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

---

## Introduction

This corrigendum contains corrections to the Ada 2012 standard [ISO/IEC 8652:2012].

The corrigendum is organized by clauses corresponding to those in the Ada 2012 standard. These clauses include wording changes to the Ada 2012 standard. Subclause headings are given for each subclause that contains a wording change. Other subclauses are omitted. For each change, a reference to the defect report(s) that prompted the wording change is included in the form [8652/0000]. The defect reports have been developed by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group to address specific questions about the Ada standard. Refer to the defect reports for details on the issues.

For each change, an *anchor* paragraph from the original Ada 2012 standard 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 Ada 2012 standard - this font for text, this font for syntax, and this font for Ada source code.

# Introduction

*Of International Standard ISO/IEC 8652:2012. Modifications of this section of that standard are found here.*

## **Replace paragraph 57.16: [8652/0117]**

- New forms of expressions are introduced. These are if expressions, case expressions, quantified expressions, and expression functions. As well as being useful for programming in general by avoiding the introduction of unnecessary assignments, they are especially valuable in conditions and invariants since they avoid the need to introduce auxiliary functions. See subclauses 4.5.7, 4.5.8, and 6.8. Membership tests are also made more flexible. See subclauses 4.4 and 4.5.2.

**by:**

- New forms of expressions are introduced. These are if expressions, case expressions, quantified expressions, expression functions, and raise expressions. As well as being useful for programming in general by avoiding the introduction of unnecessary assignments, they are especially valuable in conditions and invariants since they avoid the need to introduce auxiliary functions. See subclauses 4.5.7, 4.5.8, 6.8, and 11.3. Membership tests are also made more flexible. See subclauses 4.4 and 4.5.2.

## Section 1: General

### 1.1 Scope

**Replace paragraph 3: [8652/0118]**

The language provides rich support for real-time, concurrent programming, and includes facilities for multicore and multiprocessor programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation, and definition and use of containers.

**by:**

The language provides rich support for real-time, concurrent programming, and includes facilities for multicore and multiprocessor programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, random number generation, and definition and use of containers.

#### 1.1.2 Structure

**Replace paragraph 24: [8652/0118]**

Each section is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

**by:**

Each clause is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

## **Section 2: Lexical Elements**

No changes in this clause.

## Section 3: Declarations and Types

### 3.2.4 Subtype Predicates

**Replace paragraph 4: [8652/0119; 8652/0120]**

- For a (first) subtype defined by a derived type declaration, the predicates of the parent subtype and the progenitor subtypes apply.

**by:**

- For a (first) subtype defined by a type declaration, any predicates of parent or progenitor subtypes apply.

**Delete paragraph 6: [8652/0119]**

The *predicate* of a subtype consists of all predicate specifications that apply, and-ed together; if no predicate specifications apply, the predicate is True (in particular, the predicate of a base subtype is True).

**Replace paragraph 12: [8652/0120]**

- If a subtype is defined by a derived type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if predicate checks are enabled for at least one of the parent subtype and the progenitor subtypes;

**by:**

- If a subtype is defined by a type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if any predicate checks are enabled for parent or progenitor subtypes;

**Insert after paragraph 14: [8652/0121]**

- Otherwise, predicate checks are disabled for the given subtype.

**the new paragraphs:**

For a subtype with a directly-specified predicate aspect, the following additional language-defined aspect may be specified with an **aspect\_specification** (see 13.1.1):

**Predicate\_Failure**

This aspect shall be specified by an **expression**, which determines the action to be performed when a predicate check fails because a directly-specified predicate aspect of the subtype evaluates to **False**, as explained below.

*Name Resolution Rules*

The expected type for the **Predicate\_Failure expression** is **String**.

**Replace paragraph 17: [8652/0122]**

- a membership test whose **simple\_expression** is the current instance, and whose **membership\_choice\_list** meets the requirements for a static membership test (see 4.9);

**by:**

- a membership test whose **tested\_simple\_expression** is the current instance, and whose **membership\_choice\_list** meets the requirements for a static membership test (see 4.9);

**Replace paragraph 20: [8652/0120]**

- a call to a predefined boolean logical operator, where each operand is predicate-static;

**by:**

- a call to a predefined boolean operator **and**, **or**, **xor**, or **not**, where each operand is predicate-static;

**Insert before paragraph 30: [8652/0119]**

If predicate checks are enabled for a given subtype, then:

**the new paragraphs:**

If any of the above Legality Rules is violated in an instance of a generic unit, Program\_Error is raised at the point of the violation.

To determine whether a value *satisfies the predicates* of a subtype *S*, the following tests are performed in the following order, until one of the tests fails, in which case the predicates are not satisfied and no further tests are performed, or all of the tests succeed, in which case the predicates are satisfied:

- the value is first tested to determine whether it satisfies any constraints or any null exclusion of *S*;
- then:
  - if *S* is a first subtype, the value is tested to determine whether it satisfies the predicates of the parent and progenitor subtypes (if any) of *S* (in an arbitrary order);
  - if *S* is defined by a **subtype\_indication**, the value is tested to determine whether it satisfies the predicates of the subtype denoted by the **subtype\_mark** of the **subtype\_indication**;
- finally, if *S* is defined by a declaration to which one or more predicate specifications apply, the predicates are evaluated (in an arbitrary order) to test that all of them yield True for the given value.

**Replace paragraph 31: [8652/0121; 8652/0119]**

On every subtype conversion, the predicate of the target subtype is evaluated, and a check is performed that the predicate is True. This includes all parameter passing, except for certain parameters passed by reference, which are covered by the following rule: After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, the predicate of the subtype of the actual is evaluated, and a check is performed that the predicate is True. For an object created by an **object\_declaration** with no explicit initialization **expression**, or by an uninitialized **allocator**, if any subcomponents have **default\_expressions**, the predicate of the nominal subtype of the created object is evaluated, and a check is performed that the predicate is True. Assertions.Assertion\_Error is raised if any of these checks fail.

by:

On every subtype conversion, a check is performed that the operand satisfies the predicates of the target subtype. This includes all parameter passing, except for certain parameters passed by reference, which are covered by the following rule: After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, a check is performed that the value of the parameter satisfies the predicates of the subtype of the actual. For an object created by an **object\_declaration** with no explicit initialization **expression**, or by an uninitialized **allocator**, if any subcomponents have **default\_expressions**, a check is performed that the value of the created object satisfies the predicates of the nominal subtype.

If any of the predicate checks fail, Assertion\_Error is raised, unless the subtype whose directly-specified predicate aspect evaluated to False also has a directly-specified Predicate\_Failure aspect. In that case, the specified Predicate\_Failure **expression** is evaluated; if the evaluation of the Predicate\_Failure **expression** propagates an exception occurrence, then this occurrence is propagated for the failure of the predicate check; otherwise, Assertion\_Error is raised, with an associated message string defined by the value of the Predicate\_Failure **expression**. In the absence of such a Predicate\_Failure aspect, an implementation-defined message string is associated with the Assertion\_Error exception.

**Delete paragraph 32: [8652/0119]**

A value *satisfies* a predicate if the predicate is True for that value.

**Delete paragraph 33: [8652/0119]**

If any of the above Legality Rules is violated in an instance of a generic unit, Program\_Error is raised at the point of the violation.

**Insert after paragraph 35: [8652/0121; 8652/0119]**

6 A Static\_Predicate, like a constraint, always remains True for all objects of the subtype, except in the case of uninitialized variables and other invalid values. A Dynamic\_Predicate, on the other hand, is checked as specified above, but can become False at other times. For example, the predicate of a record subtype is not checked when a subcomponent is modified.

**the new paragraphs:**

7 No predicates apply to the base subtype of a scalar type; every value of a scalar type *T* is considered to satisfy the predicates of *T*Base.

8 Predicate\_Failure expressions are never evaluated during the evaluation of a membership test (see 4.5.2) or Valid attribute (see 13.9.2).

9 A Predicate\_Failure expression can be a raise\_expression (see 11.3).

*Examples*

```
subtype Basic_Letter is Character -- See A.3.2 for "basic letter".
with Static_Predicate => Basic_Letter in 'A'..'Z' | 'a'..'z' | 'Æ' | 'æ' |
'D' | 'ð' | 'Ð' | 'þ' | 'ß';

subtype Even_Integer is Integer
with Dynamic_Predicate => Even_Integer mod 2 = 0,
    Predicate_Failure => "Even_Integer must be a multiple of 2";
```

*Text\_IO (see A.10.1) could have used predicates to describe some common exceptional conditions as follows:*

```
with Ada.IO_Exceptions;
package Ada.Text_IO is

    type File_Type is limited private;

    subtype Open_File_Type is File_Type
    with Dynamic_Predicate => Is_Open (Open_File_Type),
        Predicate_Failure => raise Status_Error with "File not open";
    subtype Input_File_Type is Open_File_Type
    with Dynamic_Predicate => Mode (Input_File_Type) = In_File,
        Predicate_Failure => raise Mode_Error with "Cannot read file: " &
            Name (Input_File_Type);
    subtype Output_File_Type is Open_File_Type
    with Dynamic_Predicate => Mode (Output_File_Type) /= In_File,
        Predicate_Failure => raise Mode_Error with "Cannot write file: " &
            Name (Output_File_Type);

    ...

    function Mode (File : in Open_File_Type) return File_Mode;
    function Name (File : in Open_File_Type) return String;
    function Form (File : in Open_File_Type) return String;

    ...

    procedure Get (File : in Input_File_Type; Item : out Character);
    procedure Put (File : in Output_File_Type; Item : in Character);

    ...

    -- Similarly for all of the other input and output subprograms.
```



### 3.5 Scalar Types

**Insert after paragraph 55: [8652/0123]**

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

**the new paragraphs:**

For a prefix X that denotes an object of a scalar type (after any implicit dereference), the following attributes are defined:

X'Wide\_Wide\_Image

X'Wide\_Wide\_Image denotes the result of calling function S'Wide\_Wide\_Image with *Arg* being X, where S is the nominal subtype of X.

X'Wide\_Image

X'Wide\_Image denotes the result of calling function S'Wide\_Image with *Arg* being X, where S is the nominal subtype of X.

X'Image

X'Image denotes the result of calling function S'Image with *Arg* being X, where S is the nominal subtype of X.

#### 3.5.5 Operations of Discrete Types

**Replace paragraph 7.1: [8652/0119]**

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies any predicate of S, the following attributes are defined:

**by:**

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S, the following attributes are defined:

**Replace paragraph 7.2: [8652/0119]**

S'First\_Valid

S'First\_Valid denotes the smallest value that belongs to S and satisfies the predicate of S. The value of this attribute is of the type of S.

**by:**

S'First\_Valid

S'First\_Valid denotes the smallest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

**Replace paragraph 7.3: [8652/0119]**

S'Last\_Valid

S'Last\_Valid denotes the largest value that belongs to S and satisfies the predicate of S. The value of this attribute is of the type of S.

**by:**

S'Last\_Valid

S'Last\_Valid denotes the largest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

### 3.8.1 Variant Parts and Discrete Choices

**Replace paragraph 10.1: [8652/0119]**

- A `discrete_choice` that is a `subtype_indication` covers all values (possibly none) that belong to the subtype and that satisfy the static predicate of the subtype (see 3.2.4).

by:

- A `discrete_choice` that is a `subtype_indication` covers all values (possibly none) that belong to the subtype and that satisfy the static predicates of the subtype (see 3.2.4).

**Replace paragraph 15: [8652/0119]**

- If the discriminant is of a static constrained scalar subtype then, except within an instance of a generic unit, each non-**others** `discrete_choice` shall cover only values in that subtype that satisfy its predicate, and each value of that subtype that satisfies its predicate shall be covered by some `discrete_choice` (either explicitly or by **others**);

by:

- If the discriminant is of a static constrained scalar subtype then, except within an instance of a generic unit, each non-**others** `discrete_choice` shall cover only values in that subtype that satisfy its predicates, and each value of that subtype that satisfies its predicates shall be covered by some `discrete_choice` (either explicitly or by **others**);

### 3.9 Tagged Types and Type Extensions

**Replace paragraph 12.4: [8652/0118]**

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

by:

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

#### 3.9.3 Abstract Types and Subprograms

**Replace paragraph 6: [8652/0124]**

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

by:

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

### 3.10 Access Types

Replace paragraph 22: [8652/0118]

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

by:

```
type Frame is access Matrix; -- see 3.6
type Peripheral_Ref is not null access Peripheral; -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
-- general access-to-class-wide, see 3.9.1
```

#### 3.10.1 Incomplete Type Declarations

Replace paragraph 2.1: [8652/0125]

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

by:

An `incomplete_type_declaration` declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a `discriminant_part` appears. If the `incomplete_type_declaration` includes the reserved word **tagged**, it declares a *tagged incomplete view*. If *T* denotes a tagged incomplete view, then *T*'Class denotes a tagged incomplete view. An incomplete view of a type is a limited view of the type (see 7.5).

#### 3.10.2 Operations of Access Types

Replace paragraph 7: [8652/0126; 8652/0127]

- An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. Other than for an explicitly aliased parameter, a formal parameter of a callable entity has the same accessibility level as the master representing the invocation of the entity.

by:

- An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. Other than for an explicitly aliased parameter of a function or generic function, a formal parameter of a callable entity has the same accessibility level as the master representing the invocation of the entity.

Replace paragraph 10: [8652/0128]

- The accessibility level of an **aggregate** that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an **aggregate** is that of the innermost master that evaluates the **aggregate**.

by:

- The accessibility level of an **aggregate** that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an **aggregate** is that of the innermost master that evaluates the **aggregate**. Corresponding rules apply to a value conversion (see 4.6).

**Insert after paragraph 13.1: [8652/0129]**

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

**the new paragraphs:**

- The accessibility level of the anonymous access subtype defined by a `return_subtype_indication` that is an `access_definition` (see 6.5) is that of the result subtype of the enclosing function.

**Replace paragraph 19.2: [8652/0127]**

- Inside a return statement that applies to a function *F*, when determining whether the accessibility level of an explicitly aliased parameter of *F* is statically deeper than the level of the return object of *F*, the level of the return object is considered to be the same as that of the level of the explicitly aliased parameter; for statically comparing with the level of other entities, an explicitly aliased parameter of *F* is considered to have the accessibility level of the body of *F*.

**by:**

- Inside a return statement that applies to a function or generic function *F*, when determining whether the accessibility level of an explicitly aliased parameter of *F* is statically deeper than the level of the return object of *F*, the level of the return object is considered to be the same as that of the level of the explicitly aliased parameter; for statically comparing with the level of other entities, an explicitly aliased parameter of *F* is considered to have the accessibility level of the body of *F*.

**Replace paragraph 19.3: [8652/0127]**

- For determining whether a level is statically deeper than the level of the anonymous access type of an access result of a function, when within a return statement that applies to the function, the level of the master of the call is presumed to be the same as that of the level of the master that elaborated the function body.

**by:**

- For determining whether a level is statically deeper than the level of the anonymous access type of an access result of a function or generic function *F*, when within a return statement that applies to *F*, the level of the master of the call is presumed to be the same as that of the level of the master that elaborated the body of *F*.

**Replace paragraph 27.2: [8652/0130]**

- *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained. For the purposes of determining within a generic body whether *D* is unconstrained in any partial view, a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type.

**by:**

- *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained.

## Section 4: Names and Expressions

### 4.1.4 Attributes

Replace paragraph 9: [8652/0131]

An `attribute_reference` denotes a value, an object, a subprogram, or some other kind of program entity. For an `attribute_reference` that denotes a value or an object, if its type is scalar, then its nominal subtype is the base subtype of the type; if its type is tagged, its nominal subtype is the first subtype of the type; otherwise, its nominal subtype is a subtype of the type without any constraint or `null_exclusion`. Similarly, unless explicitly specified otherwise, for an `attribute_reference` that denotes a function, when its result type is scalar, its result subtype is the base subtype of the type, when its result type is tagged, the result subtype is the first subtype of the type, and when the result type is some other type, the result subtype is a subtype of the type without any constraint or `null_exclusion`.

by:

An `attribute_reference` denotes a value, an object, a subprogram, or some other kind of program entity. Unless explicitly specified otherwise, for an `attribute_reference` that denotes a value or an object, if its type is scalar, then its nominal subtype is the base subtype of the type; if its type is tagged, its nominal subtype is the first subtype of the type; otherwise, its nominal subtype is a subtype of the type without any constraint or `null_exclusion`. Similarly, unless explicitly specified otherwise, for an `attribute_reference` that denotes a function, when its result type is scalar, its result subtype is the base subtype of the type, when its result type is tagged, the result subtype is the first subtype of the type, and when the result type is some other type, the result subtype is a subtype of the type without any constraint or `null_exclusion`.

### 4.1.6 User-defined Indexing

Replace paragraph 4: [8652/0132]

These aspects are inherited by descendants of *T* (including the class-wide type *TClass*). The aspects shall not be overridden, but the functions they denote may be.

by:

These aspects are inherited by descendants of *T* (including the class-wide type *TClass*).

Insert after paragraph 17: [8652/0132]

When a `generalized_indexing` is interpreted as a constant (or variable) indexing, it is equivalent to a call on a prefixed view of one of the functions named by the `Constant_Indexing` (or `Variable_Indexing`) aspect of the type of the `indexable_container_object_prefix` with the given `actual_parameter_part`, and with the `indexable_container_object_prefix` as the prefix of the prefixed view.

the new paragraph:

NOTES

6 The `Constant_Indexing` and `Variable_Indexing` aspects cannot be redefined when inherited for a derived type, but the functions that they denote can be modified by overriding or overloading.

### 4.3.1 Record Aggregates

Replace paragraph 16: [8652/0133]

Each `record_component_association` other than an **others** choice with a  $\langle \rangle$  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, or all of them shall be of anonymous access types whose subtypes statically match.

by:

Each `record_component_association` other than an **others** choice with a  $\triangleleft$  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, or all of them shall be of anonymous access types whose subtypes statically match. In addition, Legality Rules are enforced separately for each associated component.

### 4.3.3 Array Aggregates

Replace paragraph 23.1: [8652/0134]

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

by:

Each `expression` in an `array_component_association` defines the value for the associated component(s). For an `array_component_association` with  $\triangleleft$ , the associated component(s) are initialized to the `Default_Component_Value` of the array type if this aspect has been specified for the array type; otherwise, they are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

## 4.4 Expressions

Replace paragraph 3: [8652/0135; 8652/0122]

```
relation ::=
    simple_expression [relational_operator simple_expression]
    | simple_expression [not] in membership_choice_list
```

by:

```
relation ::=
    simple_expression [relational_operator simple_expression]
    | tested_simple_expression [not] in membership_choice_list
    | raise_expression
```

Replace paragraph 3.2: [8652/0122]

```
membership_choice ::= choice_expression | range | subtype_mark
```

by:

```
membership_choice ::= choice_simple_expression | range | subtype_mark
```

### 4.5.2 Relational Operators and Membership Tests

Replace paragraph 3.1: [8652/0122]

If the tested type is tagged, then the `simple_expression` shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type for the `simple_expression` is the tested type. The expected type of a `choice_expression` in a `membership_choice`, and of a `simple_expression` of a `range` in a `membership_choice`, is the tested type of the membership operation.

by:

If the tested type is tagged, then the `tested_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 `tested_simple_expression` is the tested type. The expected type of a `choice_simple_expression` in a `membership_choice`, and of a `simple_expression` of a `range` in a `membership_choice`, is the tested type of the membership operation.

**Replace paragraph 4: [8652/0122]**

For a membership test, if the `simple_expression` is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

**by:**

For a membership test, if the `tested_simple_expression` is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

**Replace paragraph 4.1: [8652/0122]**

If a membership test includes one or more `choice_expressions` and the tested type of the membership test is limited, then the tested type of the membership test shall have a visible primitive equality operator.

**by:**

If a membership test includes one or more `choice_simple_expressions` and the tested type of the membership test is limited, then the tested type of the membership test shall have a visible primitive equality operator.

**Replace paragraph 9.8: [8652/0136]**

If the profile of an explicitly declared primitive equality operator of an untagged record type is type conformant with that of the corresponding predefined equality operator, the declaration shall occur before the type is frozen. In addition, if the untagged record type has a nonlimited partial view, then the declaration shall occur in the visible part of the enclosing package. 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:**

If the profile of an explicitly declared primitive equality operator of an untagged record type is type conformant with that of the corresponding predefined equality operator, the declaration shall occur before the type is frozen. 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 27: [8652/0122]**

For the evaluation of a membership test using `in` whose `membership_choice_list` has a single `membership_choice`, the `simple_expression` and the `membership_choice` are evaluated in an arbitrary order; the result is the result of the individual membership test for the `membership_choice`.

**by:**

For the evaluation of a membership test using `in` whose `membership_choice_list` has a single `membership_choice`, the `tested_simple_expression` and the `membership_choice` are evaluated in an arbitrary order; the result is the result of the individual membership test for the `membership_choice`.

**Replace paragraph 27.1: [8652/0122]**

For the evaluation of a membership test using `in` whose `membership_choice_list` has more than one `membership_choice`, the `simple_expression` of the membership test is evaluated first and the result of the operation is equivalent to that of a sequence consisting of an individual membership test on each `membership_choice` combined with the short-circuit control form `or else`.

**by:**

For the evaluation of a membership test using `in` whose `membership_choice_list` has more than one `membership_choice`, the `tested_simple_expression` of the membership test is evaluated first and the result of the operation is equivalent to that of a sequence consisting of an individual membership test on each `membership_choice` combined with the short-circuit control form `or else`.

**Replace paragraph 28.1: [8652/0122]**

- The `membership_choice` is a `choice_expression`, and the `simple_expression` is equal to the value of the `membership_choice`. If the tested type is a record type or a limited type, the test uses the primitive equality for the type; otherwise, the test uses predefined equality.

by:

- The `membership_choice` is a *choice\_simple\_expression*, and the *tested\_simple\_expression* is equal to the value of the `membership_choice`. If the tested type is a record type or a limited type, the test uses the primitive equality for the type; otherwise, the test uses predefined equality.

**Replace paragraph 28.2: [8652/0122]**

- The `membership_choice` is a range and the value of the `simple_expression` belongs to the given range.

by:

- The `membership_choice` is a range and the value of the *tested\_simple\_expression* belongs to the given range.

**Replace paragraph 29: [8652/0122; 8652/0119]**

- The `membership_choice` is a `subtype_mark`, the tested type is scalar, the value of the `simple_expression` belongs to the range of the named subtype, and the predicate of the named subtype evaluates to True.

by:

- The `membership_choice` is a `subtype_mark`, the tested type is scalar, the value of the *tested\_simple\_expression* belongs to the range of the named subtype, and the value satisfies the predicates of the named subtype.

**Replace paragraph 30: [8652/0122; 8652/0119]**

- The `membership_choice` is a `subtype_mark`, the tested type is not scalar, the value of the `simple_expression` satisfies any constraints of the named subtype, the predicate of the named subtype evaluates to True, and:

by:

- The `membership_choice` is a `subtype_mark`, the tested type is not scalar, the value of the *tested\_simple\_expression* satisfies any constraints of the named subtype, the value satisfies the predicates of the named subtype, and:

**Replace paragraph 30.1: [8652/0122]**

if the type of the `simple_expression` is class-wide, the value has a tag that identifies a type covered by the tested type;

by:

if the type of the *tested\_simple\_expression* is class-wide, the value has a tag that identifies a type covered by the tested type;

**Replace paragraph 30.2: [8652/0122]**

- if the tested type is an access type and the named subtype excludes null, the value of the `simple_expression` is not null;

by:

- if the tested type is an access type and the named subtype excludes null, the value of the *tested\_simple\_expression* is not null;

**Replace paragraph 30.3: [8652/0122]**

- if the tested type is a general access-to-object type, the type of the `simple_expression` is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type of the tested type is tagged and the `simple_expression` is nonnull, the tag of the object designated by the value of the `simple_expression` is covered by the designated type of the tested type.



by:

- if the tested type is a general access-to-object type, the type of the *tested\_simple\_expression* is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type of the tested type is tagged and the *tested\_simple\_expression* is nonnull, the tag of the object designated by the value of the *tested\_simple\_expression* is covered by the designated type of the tested type.

## 4.6 Type Conversions

**Replace paragraph 24.17: [8652/0128]**

- The accessibility level of the operand type shall not be statically deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, the accessibility level of the operand type shall not be statically deeper than that of the declaration of the stand-alone object. 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, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, the accessibility level of the operand type shall not be statically deeper than that of the declaration of the stand-alone object.

**Replace paragraph 24.21: [8652/0128]**

- 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. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

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

**Replace paragraph 51: [8652/0119]**

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. If predicate checks are enabled for the target subtype (see 3.2.4), a check is performed that the predicate of the target subtype is satisfied for the value.

by:

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. If predicate checks are enabled for the target subtype (see 3.2.4), a check is performed that the value satisfies the predicates of the target subtype.

**Replace paragraph 56: [8652/0137]**

- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which might raise *Constraint\_Error*), except if the object is of an access type and the view conversion is passed as an **out** parameter; in this latter case, the value of the operand object is used to initialize the formal parameter without checking against any constraint of the target subtype (see 6.4.1).

by:

- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which might raise `Constraint_Error`), except if the object is of an elementary type and the view conversion is passed as an **out** parameter; in this latter case, the value of the operand object may be used to initialize the formal parameter without checking against any constraint of the target subtype (as described more precisely in 6.4.1).

**Replace paragraph 57: [8652/0138]**

If an `Accessibility_Check` fails, `Program_Error` is raised. If a predicate check fails, `Assertions.Assertion_Error` is raised. Any other check associated with a conversion raises `Constraint_Error` if it fails.

by:

If an `Accessibility_Check` fails, `Program_Error` is raised. If a predicate check fails, the effect is as defined in subclause 3.2.4, "Subtype Predicates". Any other check associated with a conversion raises `Constraint_Error` if it fails.

**Insert after paragraph 58: [8652/0128]**

Conversion to a type is the same as conversion to an unconstrained subtype of the type.

**the new paragraphs:**

Evaluation of a value conversion of a composite type either creates a new anonymous object (similar to the object created by the evaluation of an **aggregate** or a function call) or yields a new view of the operand object without creating a new object:

- If the target type is a by-reference type and there is a type that is an ancestor of both the target type and the operand type then no new object is created;
- If the target type is an array type having aliased components and the operand type is an array type having unaliased components, then a new object is created;
- Otherwise, it is unspecified whether a new object is created.

If a new object is created, then the initialization of that object is an assignment operation.

## 4.7 Qualified Expressions

**Replace paragraph 4: [8652/0139]**

The evaluation of a `qualified_expression` evaluates the operand (and if of a universal type, converts it to the type determined by the `subtype_mark`) and checks that its value belongs to the subtype denoted by the `subtype_mark`. The exception `Constraint_Error` is raised if this check fails.

by:

The evaluation of a `qualified_expression` evaluates the operand (and if of a universal type, converts it to the type determined by the `subtype_mark`) and checks that its value belongs to the subtype denoted by the `subtype_mark`. The exception `Constraint_Error` is raised if this check fails. Furthermore, if predicate checks are enabled for the subtype denoted by the `subtype_mark`, a check is performed as defined in subclause 3.2.4, "Subtype Predicates" that the value satisfies the predicates of the subtype.

## 4.9 Static Expressions and Static Subtypes

**Replace paragraph 11: [8652/0122]**

- a membership test whose `simple_expression` is a static expression, and whose `membership_choice_list` consists only of `membership_choices` that are either static `choice_expressions`, static ranges, or `subtype_marks` that denote a static (scalar or string) subtype;

by:

- a membership test whose *tested\_simple\_expression* is a static expression, and whose *membership\_choice\_list* consists only of *membership\_choices* that are either static *choice\_simple\_expressions*, static *ranges*, or *subtype\_marks* that denote a static (scalar or string) subtype;

**Replace paragraph 32.6: [8652/0122]**

- a *choice\_expression* (or a *simple\_expression* of a *range* that occurs as a *membership\_choice* of a *membership\_choice\_list*) of a static membership test that is preceded in the enclosing *membership\_choice\_list* by another item whose individual membership test (see 4.5.2) statically yields True.

by:

- a *choice\_simple\_expression* (or a *simple\_expression* of a *range* that occurs as a *membership\_choice* of a *membership\_choice\_list*) of a static membership test that is preceded in the enclosing *membership\_choice\_list* by another item whose individual membership test (see 4.5.2) statically yields True.

#### 4.9.1 Statically Matching Constraints and Subtypes

**Replace paragraph 10: [8652/0119]**

- both subtypes are static, every value that satisfies the predicate of *S1* also satisfies the predicate of *S2*, and it is not the case that both types each have at least one applicable predicate specification, predicate checks are enabled (see 11.4.2) for *S2*, and predicate checks are not enabled for *S1*.

by:

- both subtypes are static, every value that satisfies the predicates of *S1* also satisfies the predicates of *S2*, and it is not the case that both types each have at least one applicable predicate specification, predicate checks are enabled (see 11.4.2) for *S2*, and predicate checks are not enabled for *S1*.

## Section 5: Statements

### 5.2 Assignment Statements

Replace paragraph 20: [8652/0118]

```
Writer := (Status => Open, Unit => Printer, Line_Count => 60); -- see 3.8.1
Next_Car.all := (72074, null); -- see 3.10.1
```

by:

```
Writer := (Status => Open, Unit => Printer, Line_Count => 60); -- see 3.8.1
Next.all := (72074, null, Head); -- see 3.10.1
```

### 5.4 Case Statements

Replace paragraph 7: [8652/0119]

- If the *selecting\_expression* is a name (including a *type\_conversion*, *qualified\_expression*, or *function\_call*) having a static and constrained nominal subtype, then each non-**others** *discrete\_choice* shall cover only values in that subtype that satisfy its predicate (see 3.2.4), and each value of that subtype that satisfies its predicate shall be covered by some *discrete\_choice* (either explicitly or by **others**).

by:

- If the *selecting\_expression* is a name (including a *type\_conversion*, *qualified\_expression*, or *function\_call*) having a static and constrained nominal subtype, then each non-**others** *discrete\_choice* shall cover only values in that subtype that satisfy its predicates (see 3.2.4), and each value of that subtype that satisfies its predicates shall be covered by some *discrete\_choice* (either explicitly or by **others**).

### 5.5 Loop Statements

Replace paragraph 9: [8652/0119]

For the execution of a *loop\_statement* with the *iteration\_scheme* being **for** *loop\_parameter\_specification*, the *loop\_parameter\_specification* is first elaborated. This elaboration creates the loop parameter and elaborates the *discrete\_subtype\_definition*. If the *discrete\_subtype\_definition* defines a subtype with a null range, the execution of the *loop\_statement* is complete. Otherwise, the *sequence\_of\_statements* is executed once for each value of the discrete subtype defined by the *discrete\_subtype\_definition* that satisfies the predicate of the subtype (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

by:

For the execution of a *loop\_statement* with the *iteration\_scheme* being **for** *loop\_parameter\_specification*, the *loop\_parameter\_specification* is first elaborated. This elaboration creates the loop parameter and elaborates the *discrete\_subtype\_definition*. If the *discrete\_subtype\_definition* defines a subtype with a null range, the execution of the *loop\_statement* is complete. Otherwise, the *sequence\_of\_statements* is executed once for each value of the discrete subtype defined by the *discrete\_subtype\_definition* that satisfies the predicates of the subtype (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

## 5.5.2 Generalized Loop Iteration

**Insert after paragraph 6: [8652/0140; 8652/0141]**

In a container element iterator whose *iterable\_name* has type *T*, if the *iterable\_name* denotes a constant or the Variable\_Indexing aspect is not specified for *T*, then the Constant\_Indexing aspect shall be specified for *T*.

**the new paragraphs:**

The *iterator\_name* or *iterable\_name* of an *iterator\_specification* shall not denote a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained, unless the object is known to be constrained.

A container element iterator is illegal if the call of the default iterator function that creates the loop iterator (see below) is illegal.

A generalized iterator is illegal if the iteration cursor subtype of the *iterator\_name* is a limited type at the point of the generalized iterator. A container element iterator is illegal if the default cursor subtype of the type of the *iterable\_name* is a limited type at the point of the container element iterator.

**Insert after paragraph 13: [8652/0141]**

For a forward container element iterator, the operation First of the iterator type is called on the loop iterator, to produce the initial value for the loop cursor. If the result of calling Has\_Element on the initial value is False, then the execution of the *loop\_statement* is complete. Otherwise, the *sequence\_of\_statements* is executed with the loop parameter denoting an indexing (see 4.1.6) into the iterable container object for the loop, with the only parameter to the indexing being the current value of the loop cursor; then the Next operation of the iterator type is called with the loop iterator and the loop cursor to produce the next value to be assigned to the loop cursor. This repeats until the result of calling Has\_Element on the loop cursor is False, or until the loop is left as a consequence of a transfer of control. For a reverse container element iterator, the operations Last and Previous are called rather than First and Next. If the loop parameter is a constant (see above), then the indexing uses the default constant indexing function for the type of the iterable container object for the loop; otherwise it uses the default variable indexing function.

**the new paragraph:**

Any exception propagated by the execution of a generalized iterator or container element iterator is propagated by the immediately enclosing loop statement.

## Section 6: Subprograms

### 6.1 Subprogram Declarations

Replace paragraph 39: [8652/0118]

```
function Min_Cell(X : Link) return Cell;           -- see 3.10.1
function Next_Frame(K : Positive) return Frame;    -- see 3.10
function Dot_Product(Left, Right : Vector) return Real; -- see 3.6
```

by:

```
function Min_Cell(X : Link) return Cell;           -- see 3.10.1
function Next_Frame(K : Positive) return Frame;    -- see 3.10
function Dot_Product(Left, Right : Vector) return Real; -- see 3.6
function Find(B : aliased in out Barrel; Key : String) return Real;
                                                    -- see 4.1.5
```

#### 6.1.1 Preconditions and Postconditions

Replace paragraph 1: [8652/0142]

For a subprogram or entry, the following language-defined aspects may be specified with an `aspect_specification` (see 13.1.1):

by:

For a noninstance subprogram, a generic subprogram, or an entry, the following language-defined aspects may be specified with an `aspect_specification` (see 13.1.1):

Replace paragraph 7: [8652/0143]

Within the expression for a Pre'Class or Post'Class aspect for a primitive subprogram of a tagged type *T*, a name that denotes a formal parameter of type *T* is interpreted as having type *T*'Class. Similarly, a name that denotes a formal access parameter of type access-to-*T* is interpreted as having type access-to-*T*'Class. This ensures that the expression is well-defined for a primitive subprogram of a type descended from *T*.

by:

Within the expression for a Pre'Class or Post'Class aspect for a primitive subprogram *S* of a tagged type *T*, a name that denotes a formal parameter (or *S*'Result) of type *T* is interpreted as though it had a (notional) type *NT* that is a formal derived type whose ancestor type is *T*, with directly visible primitive operations. Similarly, a name that denotes a formal access parameter (or *S*'Result) of type access-to-*T* is interpreted as having type access-to-*NT*. The result of this interpretation is that the only operations that can be applied to such names are those defined for such a formal derived type. This ensures that the expression is well-defined for any primitive subprogram of a type descended from *T*.

Insert after paragraph 17: [8652/0144]

If a renaming of a subprogram or entry *S1* overrides an inherited subprogram *S2*, then the overriding is illegal unless each class-wide precondition expression that applies to *S1* fully conforms to some class-wide precondition expression that applies to *S2* and each class-wide precondition expression that applies to *S2* fully conforms to some class-wide precondition expression that applies to *S1*.

the new paragraphs:

Pre'Class shall not be specified for an overriding primitive subprogram of a tagged type *T* unless Pre'Class is specified for the corresponding primitive subprogram of some ancestor of *T*.

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

**Replace paragraph 18: [8652/0143; 8652/0144]**

If a Pre'Class or Post'Class aspect is specified for a primitive subprogram of a tagged type *T*, then the associated expression also applies to the corresponding primitive subprogram of each descendant of *T*.

**by:**

If a Pre'Class or Post'Class aspect is specified for a primitive subprogram *S* of a tagged type *T*, or such an aspect defaults to True, then a corresponding expression also applies to the corresponding primitive subprogram *S* of each descendant of *T*. The *corresponding expression* is constructed from the associated expression as follows:

- References to formal parameters of *S* (or to *S* itself) are replaced with references to the corresponding formal parameters of the corresponding inherited or overriding subprogram *S* (or to the corresponding subprogram *S* itself).

The primitive subprogram *S* is illegal if it is not abstract and the corresponding expression for a Pre'Class or Post'Class aspect would be illegal.

**Insert after paragraph 22: [8652/0131]**

- a *dependent\_expression* of a *case\_expression*;

**the new paragraph:**

- the predicate of a *quantified\_expression*;

**Replace paragraph 26: [8652/0131]**

X'Old

For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry. The constant is of the type of *X* and is initialized to the result of evaluating *X* (as an expression) at the point of the constant declaration. The value of X'Old in the postcondition expression is the value of this constant; the type of X'Old is the type of *X*. These implicit constant declarations occur in an arbitrary order.

**by:**

X'Old

Each X'Old in a postcondition expression that is enabled denotes a constant that is implicitly declared at the beginning of the subprogram body, entry body, or accept statement.

The implicitly declared entity denoted by each occurrence of X'Old is declared as follows:

- If *X* is of an anonymous access defined by an *access\_definition A* then

*X'Old* : **constant** *A* := *X*;

- If *X* is of a specific tagged type *T* then

*anonymous* : **constant** *T*'Class := *T*'Class (*X*);

*X'Old* : *T* **renames** *T*(*anonymous*);

where the name X'Old denotes the object renaming.

- Otherwise

*X'Old* : **constant** *S* := *X*;

where *S* is the nominal subtype of *X*. This includes the case where the type of *S* is an anonymous array type or a universal type.

The nominal subtype of X'Old is as implied by the above definitions. The expected type of the prefix of an Old attribute is that of the attribute. Similarly, if an Old attribute shall resolve to be of some type, then the prefix of the attribute shall resolve to be of that type.

**Insert after paragraph 35: [8652/0131]**

The precondition checks are performed in an arbitrary order, and if any of the class-wide precondition expressions evaluate to True, it is not specified whether the other class-wide precondition expressions are evaluated. The precondition checks and any check for elaboration of the subprogram body are performed in an arbitrary order. It is not specified whether in a call on a protected operation, the checks are performed before or after starting the protected action. For an entry call, the checks are performed prior to checking whether the entry is open.

**the new paragraph:**

For a call to a task entry, the postcondition check is performed before the end of the rendezvous; for a call to a protected operation, the postcondition check is performed before the end of the protected action of the call. The postcondition check for any call is performed before the finalization of any implicitly-declared constants associated (as described above) with Old `attribute_references` but after the finalization of any other entities whose accessibility level is that of the execution of the callable construct.

**Replace paragraph 37: [8652/0143]**

For any subprogram or entry call (including dispatching calls), the checks that are performed to verify specific precondition expressions and specific and class-wide postcondition expressions are determined by those for the subprogram or entry actually invoked. Note that the class-wide postcondition expressions verified by the postcondition check that is part of a call on a primitive subprogram of type *T* includes all class-wide postcondition expressions originating in any progenitor of *T*, even if the primitive subprogram called is inherited from a type *TI* and some of the postcondition expressions do not apply to the corresponding primitive subprogram of *TI*.

**by:**

For any subprogram or entry call *S* (including dispatching calls), the checks that are performed to verify specific precondition expressions and specific and class-wide postcondition expressions are determined by those for the subprogram or entry actually invoked. Note that the class-wide postcondition expressions verified by the postcondition check that is part of a call on a primitive subprogram of type *T* includes all class-wide postcondition expressions originating in any progenitor of *T*, even if the primitive subprogram called is inherited from a type *TI* and some of the postcondition expressions do not apply to the corresponding primitive subprogram of *TI*. Any operations within a class-wide postcondition expression that were resolved as primitive operations of the (notional) formal derived type *NT*, are in the evaluation of the postcondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*. This applies to both dispatching and non-dispatching calls on *S*.

**Replace paragraph 38: [8652/0143]**

The class-wide precondition check for a call to a subprogram or entry consists solely of checking the class-wide precondition expressions that apply to the denoted callable entity (not necessarily the one that is invoked).

**by:**

The class-wide precondition check for a call to a subprogram or entry *S* consists solely of checking the class-wide precondition expressions that apply to the denoted callable entity (not necessarily to the one that is invoked). Any operations within such an expression that were resolved as primitive operations of the (notional) formal derived type *NT*, are in the evaluation of the precondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*. This applies to both dispatching and non-dispatching calls on *S*.

## 6.2 Formal Parameter Modes

**Replace paragraph 10: [8652/0128]**

A parameter of a by-reference type is passed by reference, as is an explicitly aliased parameter of any type. Each value of a by-reference type has an associated object. For a parenthesized expression,



qualified\_expression, or type\_conversion, this object is the one associated with the operand. For a conditional\_expression, this object is the one associated with the evaluated *dependent\_expression*.

by:

A parameter of a by-reference type is passed by reference, as is an explicitly aliased parameter of any type. Each value of a by-reference type has an associated object. For a parenthesized expression, qualified\_expression, or view conversion, this object is the one associated with the operand. For a value conversion, the associated object is the anonymous result object if such an object is created (see 4.6); otherwise it is the associated object of the operand. For a conditional\_expression, this object is the one associated with the evaluated *dependent\_expression*.

#### Replace paragraph 13: [8652/0118]

NOTES

6 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram\_body.

by:

NOTES

6 The mode of a formal parameter describes the direction of information transfer to or from the subprogram\_body (see 6.1).

7 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram\_body.

8 A formal parameter of mode **out** might be uninitialized at the start of the subprogram\_body (see 6.4.1).

### 6.3.1 Conformance Rules

#### Replace paragraph 10.1: [8652/0145]

- any prefixed view of a subprogram (see 4.1.3).

by:

- any prefixed view of a subprogram (see 4.1.3) without synchronization kind (see 9.5) By\_Entry or By\_Protected\_Procedure.

#### Replace paragraph 12: [8652/0145]

- The default calling convention is *protected* for a protected subprogram, and for an access-to-subprogram type with the reserved word **protected** in its definition.

by:

- The default calling convention is *protected* for a protected subprogram, a prefixed view of a subprogram with a synchronization kind of By\_Protected\_Procedure, and for an access-to-subprogram type with the reserved word **protected** in its definition.

#### Replace paragraph 13: [8652/0145]

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

by:

- The default calling convention is *entry* for an entry and a prefixed view of a subprogram with a synchronization kind of By\_Entry.

#### Insert after paragraph 20: [8652/0146]

- each constituent construct of one corresponds to an instance of the same syntactic category in the other, except that an expanded name may correspond to a *direct\_name* (or *character\_literal*) or to a different expanded name in the other; and

the new paragraph:

- corresponding defining\_identifiers occurring within the two expressions are the same; and

Replace paragraph 21: [8652/0146]

- each direct\_name, character\_literal, and selector\_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct\_name, character\_literal, or selector\_name in the other; and

by:

- each direct\_name, character\_literal, and selector\_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct\_name, character\_literal, or selector\_name in the other, or they denote corresponding declarations occurring within the two expressions; and

### 6.4.1 Parameter Associations

Insert after paragraph 6.2: [8652/0130]

- the subtype *F* shall be unconstrained, discriminated in its full view, and unconstrained in any partial view.

the new paragraph:

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

Insert after paragraph 6.25: [8652/0137]

- For a call, any default\_expression evaluated as part of the call is considered part of the call.

the new paragraph:

If the mode is **out**, the actual parameter is a view conversion, and the type of the formal parameter is an access type or a scalar type that has the Default\_Value aspect specified, then

- there shall exist a type (other than a root numeric type) that is an ancestor of both the target type and the operand type; and
- in the case of a scalar type, the type of the operand of the conversion shall have the Default\_Value aspect specified.

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

Replace paragraph 13.1: [8652/0137]

- For a scalar type that has the Default\_Value aspect specified, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any predicate;

by:

- For a scalar type that has the Default\_Value aspect specified, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any predicate, except in the following case: if the actual parameter is a view conversion and either
  - there exists no type (other than a root numeric type) that is an ancestor of both the target type and the type of the operand of the conversion; or
  - the Default\_Value aspect is unspecified for the type of the operand of the conversion

then Program\_Error is raised;

## 6.5 Return Statements

### Replace paragraph 8: [8652/0147]

If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the type of the `subtype_indication` if it is specific, or otherwise that of the value of the `expression`. A check is made that the master of the type identified by the tag of the result includes the elaboration of the master that elaborated the function body. If this check fails, `Program_Error` is raised.

### by:

If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the `expression`, unless the return object is defined by an `extended_return_object_declaration` with a `subtype_indication` that is specific, in which case it is that of the type of the `subtype_indication`. A check is made that the master of the type identified by the tag of the result includes the elaboration of the master that elaborated the function body. If this check fails, `Program_Error` is raised.

## Section 7: Packages

### 7.3.1 Private Operations

#### Replace paragraph 5.2: [8652/0148]

It is possible for there to be places where a derived type is visibly a descendant of an ancestor type, but not a descendant of even a partial view of the ancestor type, because the parent of the derived type is not visibly a descendant of the ancestor. In this case, the derived type inherits no characteristics from that ancestor, but nevertheless is within the derivation class of the ancestor for the purposes of type conversion, the "covers" relationship, and matching against a formal derived type. In this case the derived type is considered to be a *descendant* of an incomplete view of the ancestor.

by:

Furthermore, it is possible for there to be places where a derived type is known to be derived indirectly from an ancestor type, but is not a descendant of even a partial view of the ancestor type, because the parent of the derived type is not visibly a descendant of the ancestor. In this case, the derived type inherits no characteristics from that ancestor, but nevertheless is within the derivation class of the ancestor for the purposes of type conversion, the "covers" relationship, and matching against a formal derived type. In this case the derived type is effectively a *descendant* of an incomplete view of the ancestor.

### 7.3.2 Type Invariants

#### Replace paragraph 1: [8652/0149]

For a private type or private extension, the following language-defined aspects may be specified with an `aspect_specification` (see 13.1.1):

by:

For a private type, private extension, or interface, the following language-defined aspects may be specified with an `aspect_specification` (see 13.1.1):

#### Replace paragraph 3: [8652/0149]

Type\_Invariant'Class

This aspect shall be specified by an **expression**, called an *invariant expression*.

Type\_Invariant'Class may be specified on a `private_type_declaration` or a `private_extension_declaration`.

by:

Type\_Invariant'Class

This aspect shall be specified by an **expression**, called an *invariant expression*.

Type\_Invariant'Class may be specified on a `private_type_declaration`, a `private_extension_declaration`, or a `full_type_declaration` for an interface type.

#### Insert after paragraph 6: [8652/0150]

The Type\_Invariant'Class aspect shall not be specified for an untagged type. The Type\_Invariant aspect shall not be specified for an abstract type.

the new paragraph:

If a type extension occurs at a point where a private operation of some ancestor is visible and inherited, and a Type\_Invariant'Class expression applies to that ancestor, then the inherited operation shall be abstract or shall be overridden.

**Replace paragraph 10: [8652/0151; 8652/0152]**

- After successful default initialization of an object of type *T*, the check is performed on the new object;

**by:**

- After successful initialization of an object of type *T* by default (see 3.3.1), the check is performed on the new object unless the partial view of *T* has unknown discriminants;
- After successful explicit initialization of the completion of a deferred constant with a part of type *T*, if the completion is inside the immediate scope of the full view of *T*, and the deferred constant is visible outside the immediate scope of *T*, the check is performed on the part(s) of type *T*;

**Replace paragraph 17: [8652/0150]**

- is declared within the immediate scope of type *T* (or by an instance of a generic unit, and the generic is declared within the immediate scope of type *T*), and

**by:**

- is declared within the immediate scope of type *T* (or by an instance of a generic unit, and the generic is declared within the immediate scope of type *T*),

**Delete paragraph 18: [8652/0150]**

- is visible outside the immediate scope of type *T* or overrides an operation that is visible outside the immediate scope of *T*, and

**Replace paragraph 19: [8652/0150; 8652/0153]**

- has a result with a part of type *T*, or one or more parameters with a part of type *T*, or an access to variable parameter whose designated type has a part of type *T*.

**by:**

- and either:
  - has a result with a part of type *T*, or
  - has one or more **out** or **in out** parameters with a part of type *T*, or
  - has an access-to-object parameter whose designated type has a part of type *T*, or
  - is a procedure or entry that has an **in** parameter with a part of type *T*,
- and either:
  - *T* is a private type or a private extension and the subprogram or entry is visible outside the immediate scope of type *T* or overrides an inherited operation that is visible outside the immediate scope of *T*, or
  - *T* is a record extension, and the subprogram or entry is a primitive operation visible outside the immediate scope of type *T* or overrides an inherited operation that is visible outside the immediate scope of *T*.

**Insert after paragraph 20: [8652/0150]**

The check is performed on each such part of type *T*.

**the new paragraph:**

- For a view conversion to a class-wide type occurring within the immediate scope of *T*, from a specific type that is a descendant of *T* (including *T* itself), a check is performed on the part of the object that is of type *T*.

**Replace paragraph 21: [8652/0124]**

If performing checks is required by the Invariant or Invariant'Class assertion policies (see 11.4.2) in effect at the point of corresponding aspect specification applicable to a given type, then the respective invariant expression is considered *enabled*.

**by:**

If performing checks is required by the Type\_Invariant or Type\_Invariant'Class assertion policies (see 11.4.2) in effect at the point of corresponding aspect specification applicable to a given type, then the respective invariant expression is considered *enabled*.

## Section 8: Visibility Rules

### 8.1 Declarative Region

Insert after paragraph 2: [8652/0154]

- any declaration, other than that of an enumeration type, that is not a completion of a previous declaration;

the new paragraph:

- an `access_definition`;

### 8.6 The Context of Overload Resolution

Replace paragraph 9: [8652/0155]

- The expression of a `case_statement`.

by:

- The *selecting\_expression* of a `case_statement` or `case_expression`.

Insert after paragraph 17: [8652/0156]

- If a usage name appears within the declarative region of a `type_declaration` and denotes that same `type_declaration`, then it denotes the *current instance* of the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. Similarly, if a usage name appears within the declarative region of a `subtype_declaration` and denotes that same `subtype_declaration`, then it denotes the current instance of the subtype. These rules do not apply if the usage name appears within the `subtype_mark` of an `access_definition` for an access-to-object type, or within the subtype of a parameter or result of an access-to-subprogram type.

the new paragraph:

Within an `aspect_specification` for a type or subtype, the current instance represents a value of the type; it is not an object. The nominal subtype of this value is given by the subtype itself (the first subtype in the case of a `type_declaration`), prior to applying any predicate specified directly on the type or subtype. If the type or subtype is by-reference, the associated object with the value is the object associated (see 6.2) with the execution of the usage name.

Replace paragraph 27.1: [8652/0122]

Other than for the `simple_expression` of a membership test, if the expected type for a `name` or `expression` is not the same as the actual type of the `name` or `expression`, the actual type shall be convertible to the expected type (see 4.6); further, if the expected type is a named access-to-object type with designated type *D1* and the actual type is an anonymous access-to-object type with designated type *D2*, then *D1* shall cover *D2*, and the `name` or `expression` shall denote a view with an accessibility level for which the statically deeper relationship applies; in particular it shall not denote an access parameter nor a stand-alone access object.

by:

Other than for the *tested* `simple_expression` of a membership test, if the expected type for a `name` or `expression` is not the same as the actual type of the `name` or `expression`, the actual type shall be convertible to the expected type (see 4.6); further, if the expected type is a named access-to-object type with designated type *D1* and the actual type is an anonymous access-to-object type with designated type *D2*, then *D1* shall cover *D2*, and the `name` or `expression` shall denote a view with an accessibility level for which the statically deeper relationship applies; in particular it shall not denote an access parameter nor a stand-alone access object.

## Section 9: Tasks and Synchronization

### 9.3 Task Dependence - Termination of Tasks

Replace paragraph 2: [8652/0129]

- If the task is created by the evaluation of an **allocator** for a given access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

by:

- If the task is created by the evaluation of an **allocator** for a given named access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

#### 9.5.4 Requeue Statements

Insert after paragraph 5: [8652/0157]

If the requeue target has parameters, then its (prefixed) profile shall be subtype conformant with the profile of the innermost enclosing callable construct.

the new paragraphs:

Given a `requeue_statement` where the innermost enclosing callable construct is for an entry *E1*, for every specific or class-wide postcondition expression *P1* that applies to *E1*, there shall exist a postcondition expression *P2* that applies to the requeue target *E2* such that

- *P1* is fully conformant with the expression produced by replacing each reference in *P2* to a formal parameter of *E2* with a reference to the corresponding formal parameter of *E1*; and
- if *P1* is enabled, then *P2* is also enabled.

The requeue target shall not have an applicable specific or class-wide postcondition which includes an `Old attribute_reference`.

If the requeue target is declared immediately within the `task_definition` of a named task type or the `protected_definition` of a named protected type, and if the requeue statement occurs within the body of that type, and if the requeue is an external requeue, then the requeue target shall not have a specific or class-wide postcondition which includes a name denoting either the current instance of that type or any entity declared within the declaration of that type.

Replace paragraph 7: [8652/0157]

The execution of a `requeue_statement` proceeds by first evaluating the *procedure\_or\_entry\_name*, including the **prefix** identifying the target task or protected object and the **expression** identifying the entry within an entry family, if any. The `entry_body` or `accept_statement` enclosing the `requeue_statement` is then completed, finalized, and left (see 7.6.1).

by:

The execution of a `requeue_statement` proceeds by first evaluating the *procedure\_or\_entry\_name*, including the **prefix** identifying the target task or protected object and the **expression** identifying the entry within an entry family, if any. Precondition checks are then performed as for a call to the requeue target entry or subprogram. The `entry_body` or `accept_statement` enclosing the `requeue_statement` is then completed, finalized, and left (see 7.6.1).

Replace paragraph 12: [8652/0157]

If the requeue target named in the `requeue_statement` has formal parameters, then during the execution of the `accept_statement` or `entry_body` corresponding to the new entry, the formal parameters denote the



same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a `requeue_statement`; any parameter passing is implicit.

**by:**

If the requeue target named in the `requeue_statement` has formal parameters, then during the execution of the `accept_statement` or `entry_body` corresponding to the new entry and during the checking of any preconditions of the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a `requeue_statement`; any parameter passing is implicit.

## 9.7.4 Asynchronous Transfer of Control

**Insert after paragraph 13: [8652/0158]**

```
select
  delay 5.0;
  Put_Line("Calculation does not converge");
then abort
  -- This calculation should finish in 5.0 seconds;
  -- if not, it is assumed to diverge.
  Horribly_Complicated_Recursive_Function(X, Y);
end select;
```

**the new paragraph:**

Note that these examples presume that there are abort completion points within the execution of the `abortable_part`.

## Section 10: Program Structure and Compilation Issues

### 10.2.1 Elaboration Control

**Insert after paragraph 17: [8652/0159]**

A `pragma Pure` is used to specify that a library unit is *declared pure*, namely that the `Pure` aspect of the library unit is `True`; all compilation units of the library unit are declared pure. In addition, the limited view of any library package is declared pure. The declaration and body of a declared pure library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be pure. All compilation units of a declared pure library unit shall depend semantically only on declared pure `library_items`. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. Furthermore, the full view of any partial view declared in the visible part of a declared pure library unit that has any available stream attributes shall support external streaming (see 13.13.2).

**the new paragraph:**

*Erroneous Execution*

Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared at library-level in a pure package.

## Section 11: Exceptions

### 11.2 Exception Handlers

**Insert before paragraph 6: [8652/0135]**

A choice with an *exception\_name* covers the named exception. A choice with **others** covers all exceptions not named by previous choices of the same *handled\_sequence\_of\_statements*. Two choices in different *exception\_handlers* of the same *handled\_sequence\_of\_statements* shall not cover the same exception.

**the new paragraph:**

An *exception\_name* of an *exception\_choice* shall denote an exception.

### 11.3 Raise Statements and Raise Expressions

**Insert after paragraph 2: [8652/0135]**

```
raise_statement ::= raise; |
                 raise exception_name [with string_expression];
```

**the new paragraph:**

```
raise_expression ::= raise exception_name [with string_expression]
```

**Replace paragraph 3: [8652/0135]**

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.

**by:**

An *exception\_name* of a *raise\_statement* or *raise\_expression* 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.

**Replace paragraph 3.1: [8652/0135]**

The expression, if any, in a *raise\_statement*, is expected to be of type String.

**by:**

A *string\_expression* of a *raise\_statement* or *raise\_expression* is expected to be of type String.

A *raise\_expression* is expected to be of any type.

**Replace paragraph 4: [8652/0135; 8652/0160]**

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a *raise\_statement* with an *exception\_name*, the named exception is raised. If a *string\_expression* is present, the *expression* is evaluated and its value is associated with the exception occurrence. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

**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. Similarly, for the evaluation of a *raise\_expression*, the named exception is raised. In both of these cases, if a *string\_expression* is present, the *expression* is evaluated and its value is associated with the exception occurrence. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

1 If the evaluation of a *string\_expression* raises an exception, that exception is propagated rather than the one denoted by the *exception\_name* of the *raise\_statement* or *raise\_expression*.

### 11.4.1 The Package Exceptions

**Replace paragraph 10.1: [8652/0135]**

*Exception\_Message* returns the message associated with the given *Exception\_Occurrence*. For an occurrence raised by a call to *Raise\_Exception*, the message is the *Message* parameter passed to *Raise\_Exception*. For the occurrence raised by a *raise\_statement* with an *exception\_name* and a *string\_expression*, the message is the *string\_expression*. For the occurrence raised by a *raise\_statement* with an *exception\_name* but without a *string\_expression*, the message is a string giving implementation-defined information about the exception occurrence. For an occurrence originally raised in some other manner (including by the failure of a language-defined check), the message is an unspecified string. In all cases, *Exception\_Message* returns a string with lower bound 1.

**by:**

*Exception\_Message* returns the message associated with the given *Exception\_Occurrence*. For an occurrence raised by a call to *Raise\_Exception*, the message is the *Message* parameter passed to *Raise\_Exception*. For the occurrence raised by a *raise\_statement* or *raise\_expression* with an *exception\_name* and a *string\_expression*, the message is the *string\_expression*. For the occurrence raised by a *raise\_statement* or *raise\_expression* with an *exception\_name* but without a *string\_expression*, the message is a string giving implementation-defined information about the exception occurrence. For an occurrence originally raised in some other manner (including by the failure of a language-defined check), the message is an unspecified string. In all cases, *Exception\_Message* returns a string with lower bound 1.

## Section 12: Generic Units

### 12.5.1 Formal Private and Derived Types

**Replace paragraph 5.1: [8652/0161]**

The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. If the formal type is nonlimited, the actual type shall be nonlimited. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

**by:**

The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. If the formal type is nonlimited, the actual type shall be nonlimited. The actual type for a formal derived type shall be tagged if and only if the formal derived type is a private extension. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

**Insert after paragraph 15: [8652/0130]**

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:**

When enforcing Legality Rules, for the purposes of determining within a generic body whether a type is unconstrained in any partial view, a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type.

## Section 13: Representation Issues

### 13.1 Operational and Representation Aspects

**Replace paragraph 9: [8652/0162]**

A representation item that directly specifies an aspect of a subtype or type shall appear after the type is completely defined (see 3.11.1), and before the subtype or type is frozen (see 13.14). If a representation item or **aspect\_specification** is given that directly specifies an aspect of an entity, then it is illegal to give another representation item or **aspect\_specification** that directly specifies the same aspect of the entity.

**by:**

A representation item that directly specifies an aspect of a subtype or type shall appear after the type is completely defined (see 3.11.1), and before the subtype or type is frozen (see 13.14).

**Replace paragraph 9.1: [8652/0162]**

An operational item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.14). If an operational item or **aspect\_specification** is given that directly specifies an aspect of an entity, then it is illegal to give another operational item or **aspect\_specification** that directly specifies the same aspect of the entity.

**by:**

An operational item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.14).

If a representation item, operational item, or **aspect\_specification** is given that directly specifies an aspect of an entity, then it is illegal to give another representation item, operational item, or **aspect\_specification** that directly specifies the same aspect of the entity.

**Replace paragraph 10: [8652/0163]**

For an untagged derived type, it is illegal to specify a type-related representation aspect if the parent type is a by-reference type, or has any user-defined primitive subprograms.

**by:**

For an untagged derived type, it is illegal to specify a type-related representation aspect if the parent type is a by-reference type, or has any user-defined primitive subprograms. Similarly, it is illegal to specify a non-confirming type-related representation aspect for an untagged by-reference type after one or more types have been derived from it.

#### 13.1.1 Aspect Specifications

**Replace paragraph 18: [8652/0164]**

A language-defined aspect shall not be specified in an **aspect\_specification** given on a **subprogram\_body** or **subprogram\_body\_stub** that is a completion of another declaration.

**by:**

A language-defined aspect shall not be specified in an **aspect\_specification** given on a completion of a subprogram or generic subprogram.

**Replace paragraph 28: [8652/0165]**

If the **aspect\_mark** includes 'Class, then:

**by:**

If the **aspect\_mark** includes 'Class (a *class-wide aspect*), then, unless specified otherwise for a particular class-wide aspect:

## 13.2 Packed Types

### Delete paragraph 6.1: [8652/0166]

If a packed type has a component that is not of a by-reference type and has no aliased part, then such a component need not be aligned according to the Alignment of its subtype; in particular it need not be allocated on a storage element boundary.

### Insert after paragraph 7: [8652/0166]

The recommended level of support for pragma Pack is:

#### the new paragraph:

- Any component of a packed type that is of a by-reference type, that is specified as independently addressable, or that contains an aliased part, shall be aligned according to the alignment of its subtype.

### Replace paragraph 8: [8652/0166]

- For a packed record type, the components should be packed as tightly as possible subject to, the Sizes of the component subtypes, and subject to any `record_representation_clause` that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

#### by:

- For a packed record type, the components should be packed as tightly as possible subject to the above alignment requirements, the Sizes of the component subtypes, and any `record_representation_clause` that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

### Replace paragraph 9: [8652/0166]

- For a packed array type, if the Size of the component subtype is less than or equal to the word size, `Component_Size` should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size.

#### by:

- For a packed array type, if the Size of the component subtype is less than or equal to the word size, `Component_Size` should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size, unless this would violate the above alignment requirements.

## 13.3 Representation Attributes

### Replace paragraph 73.4: [8652/0167]

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved and returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X; otherwise, it returns False.

#### by:

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved. It returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X and the objects occupy at least one bit; otherwise, it returns False.

## 13.9.2 The Valid Attribute

### Replace paragraph 3: [8652/0119]

X'Valid

Yields True if and only if the object denoted by X is normal, has a valid representation, and the predicate of the nominal subtype of X evaluates to True. The value of this attribute is of the predefined type Boolean.

by:

X'Valid

Yields True if and only if the object denoted by X is normal, has a valid representation, and then, if the preceding conditions hold, the value of X also satisfies the predicates of the nominal subtype of X. The value of this attribute is of the predefined type Boolean.

### Replace paragraph 12: [8652/0119]

23 X'Valid is not considered to be a read of X; hence, it is not an error to check the validity of invalid data.

by:

23 Determining whether X is normal and has a valid representation as part of the evaluation of X'Valid is not considered to include an evaluation of X; hence, it is not an error to check the validity of an object that is invalid or abnormal. Determining whether X satisfies the predicates of its nominal subtype may include an evaluation of X, but only after it has been determined that X has a valid representation.

If X is volatile, the evaluation of X'Valid is considered a read of X.

## 13.11 Storage Management

### Replace paragraph 18: [8652/0168]

If Storage\_Size is specified for an access type, then the Storage\_Size of this pool is at least that requested, and the storage for the pool is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage\_Error is raised at the point of the **attribute\_definition\_clause**. If neither Storage\_Pool nor Storage\_Size are specified, then the meaning of Storage\_Size is implementation defined.

by:

If Storage\_Size is specified for an access type *T*, an implementation-defined pool *P* is used for the type. The Storage\_Size of *P* is at least that requested, and the storage for *P* is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage\_Error is raised at the freezing point of type *T*. The storage pool *P* is used only for allocators returning type *T* or other access types specified to use *T*'Storage\_Pool. Storage\_Error is raised by an **allocator** returning such a type if the storage space of *P* is exhausted (additional memory is not allocated).

If neither Storage\_Pool nor Storage\_Size are specified, then the meaning of Storage\_Size is implementation defined.

### 13.11.3 Default Storage Pools

#### Insert after paragraph 5: [8652/0169]

The language-defined aspect Default\_Storage\_Pool may be specified for a generic instance; it defines the default pool for access types within an instance. The expected type for the Default\_Storage\_Pool aspect is Root\_Storage\_Pool'Class. The **aspect\_definition** must be a **name** that denotes a variable. This aspect overrides any Default\_Storage\_Pool pragma that might apply to the generic unit; if the aspect is not specified, the default pool of the instance is that defined for the generic unit.



the new paragraph:

The effect of specifying aspect `Default_Storage_Pool` on an instance of a language-defined generic unit is implementation-defined.

### 13.11.6 Storage Subpool Example

Replace paragraph 11: [8652/0170]

```
type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new
  Subpools.Root_Storage_Pool_With_Subpools with record
    Storage      : Storage_Array (0 .. Pool_Size-1);
    Next_Allocation : Storage_Count := 0;
    Markers      : Subpool_Array;
    Current_Pool  : Subpool_Indexes := 1;
  end record;
```

by:

```
type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new
  Subpools.Root_Storage_Pool_With_Subpools with record
    Storage      : Storage_Array (0 .. Pool_Size);
    Next_Allocation : Storage_Count := 0;
    Markers      : Subpool_Array;
    Current_Pool  : Subpool_Indexes := 1;
  end record;
```

Replace paragraph 28: [8652/0124]

```
-- Correct the alignment if necessary:
Pool.Next_Allocation := Pool.Next_Allocation +
  ((-Pool.Next_Allocation) mod Alignment);
if Pool.Next_Allocation + Size_In_Storage_Elements >
  Pool.Pool_Size then
  raise Storage_Error; -- Out of space.
end if;
Storage_Address := Pool.Storage (Pool.Next_Allocation)'Address;
Pool.Next_Allocation :=
  Pool.Next_Allocation + Size_In_Storage_Elements;
end Allocate_From_Subpool;
```

by:

```
-- Check for the maximum supported alignment, which is the alignment of the storage area:
if Alignment > Pool.Storage'Alignment then
  raise Program_Error;
end if;
-- Correct the alignment if necessary:
Pool.Next_Allocation := Pool.Next_Allocation +
  ((-Pool.Next_Allocation) mod Alignment);
if Pool.Next_Allocation + Size_In_Storage_Elements >
  Pool.Pool_Size then
  raise Storage_Error; -- Out of space.
end if;
Storage_Address := Pool.Storage (Pool.Next_Allocation)'Address;
Pool.Next_Allocation :=
  Pool.Next_Allocation + Size_In_Storage_Elements;
end Allocate_From_Subpool;
```

### 13.13.2 Stream-Oriented Attributes

Replace paragraph 38: [8652/0165; 8652/0171]

The stream-oriented attributes may be specified for any type via an `attribute_definition_clause`. The subprogram name given in such a clause shall statically denote a subprogram that is not an abstract subprogram. Furthermore, if a stream-oriented attribute is specified for an interface type by an

`attribute_definition_clause`, the subprogram name given in the clause shall statically denote a null procedure.

**by:**

The stream-oriented attributes may be specified for any type via an `attribute_definition_clause`. Alternatively, each of the specific stream-oriented attributes may be specified using an `aspect_specification` on any `type_declaration`, with the aspect name being the corresponding attribute name. Each of the class-wide stream-oriented attributes may be specified using an `aspect_specification` for a tagged type *T* using the name of the stream-oriented attribute followed by 'Class'; such class-wide aspects do not apply to other descendants of *T*.

The subprogram name given in such an `attribute_definition_clause` or `aspect_specification` shall statically denote a subprogram that is not an abstract subprogram. Furthermore, if a specific stream-oriented attribute is specified for an interface type, the subprogram name given in the `attribute_definition_clause` or `aspect_specification` shall statically denote a null procedure.

**Replace paragraph 49: [8652/0172]**

An `attribute_reference` for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the `attribute_reference`. Furthermore, an `attribute_reference` for *T*Input is illegal if *T* is an abstract type.

**by:**

An `attribute_reference` for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the `attribute_reference`. Furthermore, an `attribute_reference` for *T*Input is illegal if *T* is an abstract type. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Unless inherited from a parent type, if any, for an untagged type having a task, protected, or explicitly limited record part, the default implementation of each of the Read, Write, Input, and Output attributes raises `Program_Error` and performs no other action.

## 13.14 Freezing Rules

**Replace paragraph 3: [8652/0173]**

The end of a `declarative_part`, `protected_body`, or a declaration of a library package or generic library package, causes *freezing* of each entity and profile declared within it, except for incomplete types. A noninstance body other than a `renames-as-body` causes freezing of each entity and profile declared before it within the same `declarative_part` that is not an incomplete type; it only causes freezing of an incomplete type if the body is within the immediate scope of the incomplete type.

**by:**

The end of a `declarative_part`, `protected_body`, or a declaration of a library package or generic library package, causes *freezing* of each entity and profile declared within it, except for incomplete types. A `proper_body`, `body_stub`, or `entry_body` causes freezing of each entity and profile declared before it within the same `declarative_part` that is not an incomplete type; it only causes freezing of an incomplete type if the body is within the immediate scope of the incomplete type.

**Insert after paragraph 5: [8652/0173; 8652/0174]**

- The occurrence of a `generic_instantiation` causes freezing, except that a `name` which is a generic actual parameter whose corresponding generic formal parameter is a formal incomplete type (see 12.5.1) does not cause freezing. In addition, if a parameter of the instantiation is defaulted, the `default_expression` or `default_name` for that parameter causes freezing.

**the new paragraphs:**

- At the occurrence of an `expression_function_declaration` that is completion, the `expression` of the expression function causes freezing.

- At the occurrence of a renames-as-body whose *callable\_entity\_name* denotes an expression function, the **expression** of the expression function causes freezing.

## Annex A: Predefined Language Environment

### Replace paragraph 3: [8652/0175; 8652/0176]

The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects.

by:

The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on any language-defined subprogram perform as specified, so long as all objects that are denoted by parameters that could be passed by reference or designated by parameters of an access type are nonoverlapping.

For the purpose of determining whether concurrent calls on text input-output subprograms are required to perform as specified above, when calling a subprogram within Text\_IO or its children that implicitly operates on one of the default input/output files, the subprogram is considered to have a parameter of Current\_Input or Current\_Output (as appropriate).

### A.4.11 String Encoding

#### Replace paragraph 54: [8652/0177]

- By a Decode function when a UTF encoded string contains an invalid encoding sequence.

by:

- By a Convert or Decode function when a UTF encoded string contains an invalid encoding sequence.

#### Replace paragraph 55: [8652/0177]

- By a Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length.

by:

- By a Convert or Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length.

### A.8.1 The Generic Package Sequential\_IO

#### Insert after paragraph 10: [8652/0178]

```
function Is_Open(File : in File_Type) return Boolean;
```

the new paragraph:

```
procedure Flush (File : in File_Type);
```

### A.8.2 File Management

#### Insert after paragraph 28: [8652/0178]

Returns True if the file is open (that is, if it is associated with an external file); otherwise, returns False.

the new paragraphs:

```
procedure Flush(File : in File_Type);
```

The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file. For a direct file, the current index is unchanged; for a stream file (see A.12.1), the current position is unchanged.

The exception `Status_Error` is propagated if the file is not open. The exception `Mode_Error` is propagated if the mode of the file is `In_File`.

## A.8.4 The Generic Package `Direct_IO`

Insert after paragraph 10: [8652/0178]

```
function Is_Open(File : in File_Type) return Boolean;
```

the new paragraph:

```
procedure Flush (File : in File_Type);
```

## A.10.3 Default Input, Output, and Error Files

Replace paragraph 21: [8652/0178]

The effect of `Flush` is the same as the corresponding subprogram in `{Sequential_IO (see A.8.2)}{Streams.Stream_IO (see A.12.1)}`. If `File` is not explicitly specified, `Current_Output` is used.

by:

The effect of `Flush` is the same as the corresponding subprogram in `Sequential_IO` (see A.8.2). If `File` is not explicitly specified, `Current_Output` is used.

## A.12.1 The Package `Streams.Stream_IO`

Replace paragraph 5: [8652/0179]

```
type File_Type is limited private;
```

by:

```
type File_Type is limited private;
pragma Preelaborable_Initialization(File_Type);
```

Replace paragraph 28: [8652/0178]

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.

by:

The subprograms given in subclause A.8.2 for the control of external files (`Create`, `Open`, `Close`, `Delete`, `Reset`, `Mode`, `Name`, `Form`, `Is_Open`, and `Flush`) are available for stream files.

Delete paragraph 28.6: [8652/0178]

The `Flush` procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file or changing the position. `Mode_Error` is propagated if the mode of the file is `In_File`.

## A.18 Containers

Insert after paragraph 5: [8652/0180]

When a formal function is used to provide an ordering for a container, it is generally required to define a strict weak ordering. A function `"<"` defines a *strict weak ordering* if it is irreflexive, asymmetric, transitive, and in addition, if  $x < y$  for any values  $x$  and  $y$ , then for all other values  $z$ ,  $(x < z)$  or  $(z < y)$ .

the new paragraphs:

*Static Semantics*

Certain subprograms declared within instances of some of the generic packages presented in this clause are said to *perform indefinite insertion*. These subprograms are those corresponding (in the sense of the copying

described in subclause 12.3) to subprograms that have formal parameters of a generic formal indefinite type and that are identified as performing indefinite insertion in the subclause defining the generic package.

If a subprogram performs indefinite insertion, then certain run-time checks are performed as part of a call to the subprogram; if any of these checks fail, then the resulting exception is propagated to the caller and the container is not modified by the call. These checks are performed for each parameter corresponding (in the sense of the copying described in 12.3) to a parameter in the corresponding generic whose type is a generic formal indefinite type. The checks performed for a given parameter are those checks explicitly specified in subclause 4.8 that would be performed as part of the evaluation of an initialized allocator whose access type is declared immediately within the instance, where:

- the value of the `qualified_expression` is that of the parameter; and
- the designated subtype of the access type is the subtype of the parameter; and
- finalization of the collection of the access type has started if and only if the finalization of the instance has started.

## A.18.2 The Generic Package Containers.Vectors

### Replace paragraph 97.1: [8652/0181]

When tampering with cursors is *prohibited* for a particular vector object  $V$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $V$ , leaving  $V$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular vector object  $V$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $V$  (or tamper with the cursors of  $V$ ), leaving  $V$  unmodified.

by:

When tampering with cursors is *prohibited* for a particular vector object  $V$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $V$ , leaving  $V$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular vector object  $V$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $V$  (or tamper with the cursors of  $V$ ), leaving  $V$  unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

### Replace paragraph 168: [8652/0124]

```

procedure Prepend (Container : in out Vector;
                    New_Item  : in      Vector;
                    Count     : in      Count_Type := 1);

```

by:

```

procedure Prepend (Container : in out Vector;
                    New_Item  : in      Vector);

```

## A.18.3 The Generic Package Containers.Doubly\_Linked\_Lists

### Replace paragraph 69.1: [8652/0181]

When tampering with cursors is *prohibited* for a particular list object  $L$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $L$ , leaving  $L$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular list object  $L$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $L$  (or tamper with the cursors of  $L$ ), leaving  $L$  unmodified.

by:

When tampering with cursors is *prohibited* for a particular list object  $L$ , `Program_Error` is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $L$ , leaving  $L$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular list object  $L$ , `Program_Error` is

propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $L$  (or tamper with the cursors of  $L$ ), leaving  $L$  unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

## A.18.4 Maps

### Replace paragraph 15.1: [8652/0181]

When tampering with cursors is *prohibited* for a particular map object  $M$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $M$ , leaving  $M$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular map object  $M$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $M$  (or tamper with the cursors of  $M$ ), leaving  $M$  unmodified.

by:

When tampering with cursors is *prohibited* for a particular map object  $M$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $M$ , leaving  $M$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular map object  $M$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $M$  (or tamper with the cursors of  $M$ ), leaving  $M$  unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

## A.18.7 Sets

### Replace paragraph 14.1: [8652/0181]

When tampering with cursors is *prohibited* for a particular set object  $S$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $S$ , leaving  $S$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular set object  $S$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $S$  (or tamper with the cursors of  $S$ ), leaving  $S$  unmodified.

by:

When tampering with cursors is *prohibited* for a particular set object  $S$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of  $S$ , leaving  $S$  unmodified. Similarly, when tampering with elements is *prohibited* for a particular set object  $S$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of  $S$  (or tamper with the cursors of  $S$ ), leaving  $S$  unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

## A.18.10 The Generic Package Containers.Multiway\_Trees

### Replace paragraph 2: [8652/0182]

A multiway tree container object manages a tree of internal *nodes*, each of which contains an element and pointers to the parent, first child, last child, next (successor) sibling, and previous (predecessor) sibling internal nodes. A cursor designates a particular node within a tree (and by extension the element contained in that node, if any). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved within the container.

by:

A multiway tree container object manages a tree of *nodes*, comprising of a *root node*, and a set of *internal nodes* each of which contains an element and pointers to the parent, first child, last child, next (successor) sibling, and previous (predecessor) sibling internal nodes. A cursor designates a particular node within a tree (and by extension the element contained in that node, if any). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved within the container.

**Replace paragraph 3: [8652/0182]**

A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). There is a special node, the *root*, which is always present and has neither an associated element value nor any parent node. The root node provides a place to add nodes to an otherwise empty tree and represents the base of the tree.

**by:**

A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). The root node is always present and has neither an associated element value nor any parent node; it has pointers to its first child and its last child, if any. The root node provides a place to add nodes to an otherwise empty tree and represents the base of the tree.

**Replace paragraph 90: [8652/0181]**

When tampering with cursors is *prohibited* for a particular tree object *T*, Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *T*, leaving *T* unmodified. Similarly, when tampering with elements is *prohibited* for a particular tree object *T*, Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *T* (or tamper with the cursors of *T*), leaving *T* unmodified.

**by:**

When tampering with cursors is *prohibited* for a particular tree object *T*, Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *T*, leaving *T* unmodified. Similarly, when tampering with elements is *prohibited* for a particular tree object *T*, Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *T* (or tamper with the cursors of *T*), leaving *T* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

**Replace paragraph 153: [8652/0183]**

Iterate calls Process.all with a cursor that designates each element in Container, starting with the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.all. Any exception raised by Process.all is propagated.

**by:**

Iterate calls Process.all with a cursor that designates each element in Container, starting from the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.all. Any exception raised by Process.all is propagated.

**Replace paragraph 155: [8652/0183]**

If Position equals No\_Element, then Constraint\_Error is propagated. Otherwise, Iterate\_Subtree calls Process.all with a cursor that designates each element in the subtree rooted by the node designated by Position, starting with the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.all. Any exception raised by Process.all is propagated.

**by:**

If Position equals No\_Element, then Constraint\_Error is propagated. Otherwise, Iterate\_Subtree calls Process.all with a cursor that designates each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.all. Any exception raised by Process.all is propagated.

**Replace paragraph 157: [8652/0183]**

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in



the `sequence_of_statements` of the `loop_statement` whose `iterator_specification` denotes this object). The iterator object needs finalization.

by:

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting from the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the `sequence_of_statements` of the `loop_statement` whose `iterator_specification` denotes this object). The iterator object needs finalization.

**Replace paragraph 159: [8652/0183]**

If Position equals No\_Element, then Constraint\_Error is propagated. Otherwise, Iterate\_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position, starting with the node designated by Position and proceeding in a depth-first order. If Position equals No\_Element, then Constraint\_Error is propagated. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the `sequence_of_statements` of the `loop_statement` whose `iterator_specification` denotes this object). The iterator object needs finalization.

by:

If Position equals No\_Element, then Constraint\_Error is propagated. Otherwise, Iterate\_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order. If Position equals No\_Element, then Constraint\_Error is propagated. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the `sequence_of_statements` of the `loop_statement` whose `iterator_specification` denotes this object). The iterator object needs finalization.

### A.18.11 The Generic Package Containers.Indefinite\_Vectors

**Insert after paragraph 8: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element may be constrained even if Element\_Type is unconstrained.

the new paragraph:

- The operations "&", Append, Insert, Prepend, Replace\_Element, and To\_Vector that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### A.18.12 The Generic Package Containers.Indefinite\_Doubly\_Linked\_Lists

**Insert after paragraph 7: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element may be constrained even if Element\_Type is unconstrained.

the new paragraph:

- The operations Append, Insert, Prepend, and Replace\_Element that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### **A.18.13 The Generic Package Containers.Indefinite\_Hashed\_Maps**

**Insert after paragraph 8: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element may be constrained even if Element\_Type is unconstrained.

**the new paragraph:**

- The operations Include, Insert, Replace, and Replace\_Element that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### **A.18.14 The Generic Package Containers.Indefinite\_Ordered\_Maps**

**Insert after paragraph 8: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element may be constrained even if Element\_Type is unconstrained.

**the new paragraph:**

- The operations Include, Insert, Replace, and Replace\_Element that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### **A.18.15 The Generic Package Containers.Indefinite\_Hashed\_Sets**

**Insert after paragraph 4: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element\_Preserving\_Key may be constrained even if Element\_Type is unconstrained.

**the new paragraph:**

- The operations Include, Insert, Replace, Replace\_Element, and To\_Set that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### **A.18.16 The Generic Package Containers.Indefinite\_Ordered\_Sets**

**Insert after paragraph 4: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element\_Preserving\_Key may be constrained even if Element\_Type is unconstrained.

**the new paragraph:**

- The operations Include, Insert, Replace, Replace\_Element, and To\_Set that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### **A.18.17 The Generic Package Containers.Indefinite\_Multiway\_Trees**

**Insert after paragraph 7: [8652/0180]**

- The actual Element parameter of access subprogram Process of Update\_Element may be constrained even if Element\_Type is unconstrained.

**the new paragraph:**

- The operations Append\_Child, Insert\_Child, Prepend\_Child, and Replace\_Element that have a formal parameter of type Element\_Type perform indefinite insertion (see A.18).

### A.18.18 The Generic Package Containers.Indefinite\_Holders

#### Replace paragraph 35: [8652/0181]

When tampering with the element is *prohibited* for a particular holder object  $H$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the element of  $H$ , leaving  $H$  unmodified.

by:

When tampering with the element is *prohibited* for a particular holder object  $H$ , Program\_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the element of  $H$ , leaving  $H$  unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

#### Replace paragraph 39: [8652/0180]

Returns a nonempty holder containing an element initialized to New\_Item.

by:

Returns a nonempty holder containing an element initialized to New\_Item. To\_Holder performs indefinite insertion (see A.18).

#### Replace paragraph 47: [8652/0180]

Replace\_Element assigns the value New\_Item into Container, replacing any preexisting content of Container. Container is not empty after a successful call to Replace\_Element.

by:

Replace\_Element assigns the value New\_Item into Container, replacing any preexisting content of Container; Replace\_Element performs indefinite insertion (see A.18). Container is not empty after a successful call to Replace\_Element.

### A.18.25 The Generic Package Containers.Bounded\_Multiway\_Trees

#### Replace paragraph 10: [8652/0118]

```
function Copy (Source : Tree; Capacity : Count_Type := 0)
  return List;
```

by:

```
function Copy (Source : Tree; Capacity : Count_Type := 0)
  return Tree;
```

### A.18.26 Array Sorting

#### Replace paragraph 9.2: [8652/0118]

```
generic
  type Index_Type is (<>);
  with function Before (Left, Right : Index_Type) return Boolean;
  with procedure Swap (Left, Right : Index_Type);
procedure Ada.Containers.Generic_Sort
  (First, Last : Index_Type'Base);
pragma Pure(Ada.Containers.Generic_Sort);
```

by:

```
generic
  type Index_Type is (<>);
  with function Before (Left, Right : Index_Type) return Boolean;
  with procedure Swap (Left, Right : in Index_Type);
procedure Ada.Containers.Generic_Sort
  (First, Last : Index_Type'Base);
```

```
pragma Pure(Ada.Containers.Generic_Sort);
```

### A.18.32 Example of Container Use

Replace paragraph 29: [8652/0124]

```
for C in G (Next).Iterate loop
  declare
    E : Edge renames G (Next) (C).all;
  begin
    if not Reached(E.To) then
      ...
    end if;
  end;
end loop;
```

by:

```
for C in G (Next).Iterate loop
  declare
    E : Edge renames G (Next) (C);
  begin
    if not Reached(E.To) then
      ...
    end if;
  end;
end loop;
```

Replace paragraph 31: [8652/0124]

```
declare
  L : Adjacency_Lists.List renames G (Next);
  C : Adjacency_Lists.Cursor := L.First;
begin
  while Has_Element (C) loop
    declare
      E : Edge renames L(C).all;
    begin
      if not Reached(E.To) then
        ...
      end if;
    end;
    C := L.Next (C);
  end loop;
end;
```

by:

```
declare
  L : Adjacency_Lists.List renames G (Next);
  C : Adjacency_Lists.Cursor := L.First;
begin
  while Has_Element (C) loop
    declare
      E : Edge renames L(C);
    begin
      if not Reached(E.To) then
        ...
      end if;
    end;
    C := L.Next (C);
  end loop;
end;
```

## A.19 The Package Locales

Replace paragraph 4: [8652/0184]

```
type Language_Code is array (1 .. 3) of Character range 'a' .. 'z';  
type Country_Code is array (1 .. 2) of Character range 'A' .. 'Z';
```

by:

```
type Language_Code is new String (1 .. 3)  
  with Dynamic_Predicate => (for all E of Language_Code => E in 'a' ..  
'z');  
type Country_Code is new String (1 .. 2)  
  with Dynamic_Predicate => (for all E of Country_Code => E in 'A' ..  
'Z');
```

## Annex B: Interface to Other Languages

### B.1 Interfacing Aspects

**Replace paragraph 50: [8652/0124]**

*Example of interfacing pragmas:*

**by:**

*Example of interfacing aspects:*

### B.3 Interfacing with C and C++

**Replace paragraph 1: [8652/0185]**

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, and support for specifying the Convention aspect with *convention\_identifiers* C and 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, and support for specifying the Convention aspect with *convention\_identifiers* C, C\_Pass\_By\_Copy, and any of the C\_Variadic\_*n* conventions described below.

**Insert after paragraph 60.15: [8652/0185]**

If a type is C\_Pass\_By\_Copy-compatible, then it is also C-compatible.

**the new paragraph:**

The identifiers C\_Variadic\_0, C\_Variadic\_1, C\_Variadic\_2, and so on are *convention\_identifiers*. These conventions are said to be *C\_Variadic*. The convention C\_Variadic\_*n* is the calling convention for a variadic C function taking *n* fixed parameters and then a variable number of additional parameters. The C\_Variadic\_*n* convention shall only be specified as the convention aspect for a subprogram, or for an access-to-subprogram type, having at least *n* parameters. A type is compatible with a C\_Variadic convention if and only if the type is C-compatible.

**Replace paragraph 75: [8652/0185]**

A C function that takes a variable number of arguments can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

**by:**

A variadic C function can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

## Annex C: Systems Programming

### C.5 Aspect Discard\_Names

**Replace the title:** [8652/0186]

Pragma Discard\_Names

**by:**

Aspect Discard\_Names

**Replace paragraph 1:** [8652/0186]

A pragma Discard\_Names may be used to request a reduction in storage used for the names of certain entities.

**by:**

Specifying the aspect Discard\_Names can be used to request a reduction in storage used for the names of entities with runtime name text.

*Static Semantics*

An entity with *runtime name text* is a nonderived enumeration first subtype, a tagged first subtype, or an exception.

For an entity with runtime name text, the following language-defined representation aspect may be specified:

Discard\_Names

The type of aspect Discard\_Names is Boolean. If directly specified, the **aspect\_definition** shall be a static expression. If not specified (including by inheritance), the aspect is False.

**Replace paragraph 5:** [8652/0186]

The **local\_name** (if present) shall denote a nonderived enumeration [first] subtype, a tagged [first] subtype, or an exception. The pragma applies to the type or exception. Without a **local\_name**, the pragma applies to all such entities declared after the pragma, within the same declarative region. Alternatively, the pragma can be used as a configuration pragma. If the pragma applies to a type, then it applies also to all descendants of the type.

**by:**

The **local\_name** (if present) shall denote an entity with runtime name text. The pragma specifies that the aspect Discard\_Names for the type or exception has the value True. Without a **local\_name**, the pragma specifies that all entities with runtime name text declared after the pragma, within the same declarative region have the value True for aspect Discard\_Names. Alternatively, the pragma can be used as a configuration pragma. If the configuration pragma Discard\_Names applies to a compilation unit, all entities with runtime name text declared in the compilation unit have the value True for the aspect Discard\_Names.

**Replace paragraph 7:** [8652/0186]

If the pragma applies to an enumeration type, then the semantics of the Wide\_Wide\_Image and Wide\_Wide\_Value attributes are implementation defined for that type; the semantics of Image, Wide\_Image, Value, and Wide\_Value are still defined in terms of Wide\_Wide\_Image and Wide\_Wide\_Value. In addition, the semantics of Text\_IO Enumeration\_IO are implementation defined. If the pragma applies to a tagged type, then the semantics of the Tags.Wide\_Wide\_Expanded\_Name function are implementation defined for that type; the semantics of Tags.Expanded\_Name and Tags.Wide\_Expanded\_Name are still defined in terms of Tags.Wide\_Wide\_Expanded\_Name. If the pragma applies to an exception, then the semantics of the Exceptions.Wide\_Wide\_Exception\_Name function are implementation defined for that exception; the semantics of Exceptions.Exception\_Name and Exceptions.Wide\_Exception\_Name are still defined in terms of Exceptions.Wide\_Wide\_Exception\_Name.

by:

If the aspect Discard\_Names is True for an enumeration type, then the semantics of the Wide\_Wide\_Image and Wide\_Wide\_Value attributes are implementation defined for that type; the semantics of Image, Wide\_Image, Value, and Wide\_Value are still defined in terms of Wide\_Wide\_Image and Wide\_Wide\_Value. In addition, the semantics of Text\_IO Enumeration\_IO are implementation defined. If the aspect Discard\_Names is True for a tagged type, then the semantics of the Tags.Wide\_Wide\_Expanded\_Name function are implementation defined for that type; the semantics of Tags.Expanded\_Name and Tags.Wide\_Expanded\_Name are still defined in terms of Tags.Wide\_Wide\_Expanded\_Name. If the aspect Discard\_Names is True for an exception, then the semantics of the Exceptions.Wide\_Wide\_Exception\_Name function are implementation defined for that exception; the semantics of Exceptions.Exception\_Name and Exceptions.Wide\_Exception\_Name are still defined in terms of Exceptions.Wide\_Wide\_Exception\_Name.

**Replace paragraph 8: [8652/0186]**

If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.

by:

If the aspect Discard\_Names is True for an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.

## C.6 Shared Variable Control

**Replace paragraph 8.1: [8652/0166]**

When True, the aspects Independent and Independent\_Components *specify as independently addressable* the named object or component(s), or in the case of a type, all objects or components of that type. All atomic objects are considered to be specified as independently addressable.

by:

When True, the aspects Independent and Independent\_Components *specify as independently addressable* the named object or component(s), or in the case of a type, all objects or components of that type. All atomic objects and aliased objects are considered to be specified as independently addressable.

**Replace paragraph 10: [8652/0166]**

It is illegal to specify either of the aspects Atomic or Atomic\_Components to have the value True for an object or type if the implementation cannot support the indivisible reads and updates required by the aspect (see below).

by:

It is illegal to specify either of the aspects Atomic or Atomic\_Components to have the value True for an object or type if the implementation cannot support the indivisible and independent reads and updates required by the aspect (see below).

**Replace paragraph 11: [8652/0166]**

It is illegal to specify the Size attribute of an atomic object, the Component\_Size attribute for an array type with atomic components, or the layout attributes of an atomic component, in a way that prevents the implementation from performing the required indivisible reads and updates.

by:

It is illegal to specify the Size attribute of an atomic object, the Component\_Size attribute for an array type with atomic components, or the layout attributes of an atomic component, in a way that prevents the implementation from performing the required indivisible and independent reads and updates.



**Delete paragraph 21: [8652/0166]**

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.

**Insert after paragraph 24: [8652/0166]**

NOTES:

9 An imported volatile or atomic constant behaves as a constant (i.e. read-only) with respect to other parts of the Ada program, but can still be modified by an "external source."

**the new paragraph:**

10 Specifying the Pack aspect cannot override the effect of specifying an Atomic or Atomic\_Components aspect.

## Annex D: Real-Time Systems

### D.1 Task Priorities

**Replace paragraph 17: [8652/0187]**

The **expression** specified for the Priority or Interrupt\_Priority aspect of a task is evaluated for each task object (see 9.1). For the Priority aspect, the value of the **expression** is converted to the subtype Priority; for the Interrupt\_Priority aspect, this value is converted to the subtype Any\_Priority. The priority value is then associated with the task object whose task declaration specifies the aspect.

**by:**

The **expression** specified for the Priority or Interrupt\_Priority aspect of a task type is evaluated each time an object of the task type is created (see 9.1). For the Priority aspect, the value of the **expression** is converted to the subtype Priority; for the Interrupt\_Priority aspect, this value is converted to the subtype Any\_Priority. The priority value is then associated with the task object.

### D.3 Priority Ceiling Locking

**Replace paragraph 10: [8652/0188]**

- If an Interrupt\_Handler or Attach\_Handler aspect (see C.3.1) is specified for a protected subprogram of a protected type that does not have the Interrupt\_Priority aspect specified, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt\_Priority.

**by:**

- If an Interrupt\_Handler or Attach\_Handler aspect (see C.3.1) is specified for a protected subprogram of a protected type that does not have either the Priority or Interrupt\_Priority aspect specified, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt\_Priority.

### D.7 Tasking Restrictions

**Insert after paragraph 10: [8652/0189]**

No\_Dynamic\_Attachment

There is no use of a **name** denoting any of the operations defined in package Interrupts (Is\_Reserved, Is\_Attached, Current\_Handler, Attach\_Handler, Exchange\_Handler, Detach\_Handler, and Reference).

**the new paragraph:**

No\_Dynamic\_CPU\_Assignment

No task has the CPU aspect specified to be a non-static expression. Each task (including the environment task) that has the CPU aspect specified as Not\_A\_Specific\_CPU will be assigned to a particular implementation-defined CPU. The same is true for the environment task when the CPU aspect is not specified. Any other task without a CPU aspect will activate and execute on the same processor as its activating task.

**Insert after paragraph 10.8: [8652/0190]**

No\_Specific\_Termination\_Handlers

There is no use of a **name** denoting the Set\_Specific\_Handler and Specific\_Handler subprograms in Task\_Termination.

**the new paragraph:**

No\_Tasks\_Unassigned\_To\_CPU

The CPU aspect is specified for the environment task. No CPU aspect is specified to be statically equal to Not\_A\_Specific\_CPU. If aspect CPU is specified (dynamically) to the value Not\_A\_Specific\_CPU, then Program\_Error is raised. If Set\_CPU or Delay\_Until\_And\_Set\_CPU are called with the CPU parameter equal to Not\_A\_Specific\_CPU, then Program\_Error is raised.

## D.13 The Ravenscar Profile

Replace paragraph 6: [8652/0189; 8652/0191]

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
    No_Abort_Statements,
    No_Dynamic_Attachment,
    No_Dynamic_Priorities,
    No_Implicit_Heap_Allocations,
    No_Local_Protected_Objects,
    No_Local_Timing_Events,
    No_Protected_Type_Allocators,
    No_Relative_Delay,
    No_Requeue_Statements,
    No_Select_Statements,
    No_Specific_Termination_Handlers,
    No_Task_Allocators,
    No_Task_Hierarchy,
    No_Task_Termination,
    Simple_Barriers,
    Max_Entry_Queue_Length => 1,
    Max_Protected_Entries => 1,
    Max_Task_Entries => 0,
    No_Dependence => Ada.Asynchronous_Task_Control,
    No_Dependence => Ada.Calendar,
    No_Dependence => Ada.Execution_Time.Group_Budgets,
    No_Dependence => Ada.Execution_Time.Timers,
    No_Dependence => Ada.Task_Attributes,
    No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

by:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
    No_Abort_Statements,
    No_Dynamic_Attachment,
    No_Dynamic_CPU_Assignment,
    No_Dynamic_Priorities,
    No_Implicit_Heap_Allocations,
    No_Local_Protected_Objects,
    No_Local_Timing_Events,
    No_Protected_Type_Allocators,
    No_Relative_Delay,
    No_Requeue_Statements,
    No_Select_Statements,
    No_Specific_Termination_Handlers,
    No_Task_Allocators,
    No_Task_Hierarchy,
    No_Task_Termination,
    Simple_Barriers,
    Max_Entry_Queue_Length => 1,
    Max_Protected_Entries => 1,
    Max_Task_Entries => 0,
    No_Dependence => Ada.Asynchronous_Task_Control,
    No_Dependence => Ada.Calendar,
    No_Dependence => Ada.Execution_Time.Group_Budgets,
    No_Dependence => Ada.Execution_Time.Timers,
```

```
No_Dependence => Ada.Synchronous_Barriers,  
No_Dependence => Ada.Task_Attributes,  
No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

**Delete paragraph 8: [8652/0189]**

A task shall only be on the ready queues of one processor, and the processor to which a task belongs shall be defined statically. Whenever a task running on a processor reaches a task dispatching point, it goes back to the ready queues of the same processor. A task with a CPU value of Not\_A\_Specific\_CPU will execute on an implementation defined processor. A task without a CPU aspect will activate and execute on the same processor as its activating task.

**Insert after paragraph 10: [8652/0189]**

NOTES

42 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.

**the new paragraphs:**

43 When the Ravenscar profile is in effect (via the effect of the No\_Dynamic\_CPU\_Assignment restriction), all of the tasks in the partition will execute on a single CPU unless the programmer explicitly uses aspect CPU to specify the CPU assignments for tasks. The use of multiple CPUs requires care, as many guarantees of single CPU scheduling no longer apply.

44 It is not recommended to specify the CPU of a task to be Not\_A\_Specific\_CPU when the Ravenscar profile is in effect. How a partition executes strongly depends on the assignment of tasks to CPUs.

## D.16 Multiprocessor Implementation

**Replace paragraph 9: [8652/0187]**

The expression specified for the CPU aspect of a task is evaluated for each task object (see 9.1). The CPU value is then associated with the task object whose task declaration specifies the aspect.

**by:**

The expression specified for the CPU aspect of a task type is evaluated each time an object of the task type is created (see 9.1). The CPU value is then associated with the task object.

### D.16.1 Multiprocessor Dispatching Domains

**Replace paragraph 7: [8652/0192]**

```
function Create (First, Last : CPU) return Dispatching_Domain;
```

**by:**

```
function Create (First : CPU; Last : CPU_Range) return Dispatching_Domain;
```

**Replace paragraph 9: [8652/0192]**

```
function Get_Last_CPU (Domain : Dispatching_Domain) return CPU;
```

**by:**

```
function Get_Last_CPU (Domain : Dispatching_Domain) return CPU_Range;
```

```
type CPU_Set is array(CPU range <>) of Boolean;
```

```
function Create (Set : CPU_Set) return Dispatching_Domain;
```

```
function Get_CPU_Set (Domain : Dispatching_Domain) return CPU_Set;
```

**Replace paragraph 16: [8652/0193]**

The type Dispatching\_Domain represents a series of processors on which a task may execute. Each processor is contained within exactly one Dispatching\_Domain. System\_Dispatching\_Domain contains the processor or

processors on which the environment task executes. At program start-up all processors are contained within System\_Dispatching\_Domain.

**by:**

A *dispatching domain* represents a set of processors on which a task may execute. Each processor is contained within exactly one dispatching domain. An object of type Dispatching\_Domain identifies a dispatching domain. System\_Dispatching\_Domain identifies a domain that contains the processor or processors on which the environment task executes. At program start-up all processors are contained within this domain.

**Replace paragraph 20: [8652/0192]**

The expression specified for the Dispatching\_Domain aspect of a task is evaluated for each task object (see 9.1). The Dispatching\_Domain value is then associated with the task object whose task declaration specifies the aspect.

**by:**

The expression specified for the Dispatching\_Domain aspect of a task type is evaluated each time an object of the task type is created (see 9.1). If the identified dispatching domain is empty, then Dispatching\_Domain\_Error is raised; otherwise the newly created task is assigned to the domain identified by the value of the expression.

**Replace paragraph 22: [8652/0193]**

If both Dispatching\_Domain and CPU are specified for a task, and the CPU value is not contained within the range of processors for the domain (and is not Not\_A\_Specific\_CPU), the activation of the task is defined to have failed, and it becomes a completed task (see 9.2).

**by:**

If both the dispatching domain and CPU are specified for a task, and the CPU value is not contained within the set of processors for the domain (and is not Not\_A\_Specific\_CPU), the activation of the task is defined to have failed, and it becomes a completed task (see 9.2).

**Replace paragraph 23: [8652/0192]**

The function Create creates and returns a Dispatching\_Domain containing all the processors in the range First .. Last. These processors are removed from System\_Dispatching\_Domain. A call of Create will raise Dispatching\_Domain\_Error if any designated processor is not currently in System\_Dispatching\_Domain, or if the system cannot support a distinct domain over the processors identified, or if a processor has a task assigned to it, or if the allocation would leave System\_Dispatching\_Domain empty. A call of Create will raise Dispatching\_Domain\_Error if the calling task is not the environment task, or if Create is called after the call to the main subprogram.

**by:**

The function Create with First and Last parameters creates and returns a dispatching domain containing all the processors in the range First .. Last. The function Create with a Set parameter creates and returns a dispatching domain containing the processors for which Set(I) is True. These processors are removed from System\_Dispatching\_Domain. A call of Create will raise Dispatching\_Domain\_Error if any designated processor is not currently in System\_Dispatching\_Domain, or if the system cannot support a distinct domain over the processors identified, or if a processor has a task assigned to it, or if the allocation would leave System\_Dispatching\_Domain empty. A call of Create will raise Dispatching\_Domain\_Error if the calling task is not the environment task, or if Create is called after the call to the main subprogram.

**Replace paragraph 24: [8652/0192]**

The function Get\_First\_CPU returns the first CPU in Domain; Get\_Last\_CPU returns the last one.

**by:**

The function Get\_First\_CPU returns the first CPU in Domain, or CPU'First if Domain is empty; Get\_Last\_CPU returns the last CPU in Domain, or CPU\_Range'First if Domain is empty. The function

Get\_CPU\_Set(D) returns an array whose low bound is Get\_First\_CPU(D), whose high bound is Get\_Last\_CPU(D), with True values in the Set corresponding to the CPUs that are in the given Domain.

**Replace paragraph 25: [8652/0193]**

The function Get\_Dispatching\_Domain returns the Dispatching\_Domain on which the task is assigned.

**by:**

The function Get\_Dispatching\_Domain returns the dispatching domain on which the task is assigned.

**Replace paragraph 26: [8652/0192]**

A call of the procedure Assign\_Task assigns task T to the CPU within Dispatching\_Domain Domain. Task T can now execute only on CPU unless CPU designates Not\_A\_Specific\_CPU, in which case it can execute on any processor within Domain. The exception Dispatching\_Domain\_Error is propagated if T is already assigned to a Dispatching\_Domain other than System\_Dispatching\_Domain, or if CPU is not one of the processors of Domain (and is not Not\_A\_Specific\_CPU). A call of Assign\_Task is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current\_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical. Assigning a task to System\_Dispatching\_Domain that is already assigned to that domain has no effect.

**by:**

A call of the procedure Assign\_Task assigns task T to the CPU within the dispatching domain Domain. Task T can now execute only on CPU, unless CPU designates Not\_A\_Specific\_CPU in which case it can execute on any processor within Domain. The exception Dispatching\_Domain\_Error is propagated if Domain is empty, T is already assigned to a dispatching domain other than System\_Dispatching\_Domain, or if CPU is not one of the processors of Domain (and is not Not\_A\_Specific\_CPU). A call of Assign\_Task is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current\_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical. Assigning a task already assigned to System\_Dispatching\_Domain to that domain has no effect.

**Replace paragraph 27: [8652/0193]**

A call of procedure Set\_CPU assigns task T to the CPU. Task T can now execute only on CPU, unless CPU designates Not\_A\_Specific\_CPU, in which case it can execute on any processor within its Dispatching\_Domain. The exception Dispatching\_Domain\_Error is propagated if CPU is not one of the processors of the Dispatching\_Domain on which T is assigned (and is not Not\_A\_Specific\_CPU). A call of Set\_CPU is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current\_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical.

**by:**

A call of procedure Set\_CPU assigns task T to the CPU. Task T can now execute only on CPU, unless CPU designates Not\_A\_Specific\_CPU, in which case it can execute on any processor within its dispatching domain. The exception Dispatching\_Domain\_Error is propagated if CPU is not one of the processors of the dispatching domain on which T is assigned (and is not Not\_A\_Specific\_CPU). A call of Set\_CPU is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current\_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical.

**Replace paragraph 29: [8652/0193]**

A call of Delay\_Until\_And\_Set\_CPU delays the calling task for the designated time and then assigns the task to the specified processor when the delay expires. The exception Dispatching\_Domain\_Error is propagated if P is not one of the processors of the calling task's Dispatching\_Domain (and is not Not\_A\_Specific\_CPU).

**by:**

A call of Delay\_Until\_And\_Set\_CPU delays the calling task for the designated time and then assigns the task to the specified processor when the delay expires. The exception Dispatching\_Domain\_Error is propagated if P is not one of the processors of the calling task's dispatching domain (and is not Not\_A\_Specific\_CPU).

**Insert after paragraph 30: [8652/0194]**

The implementation shall perform the operations Assign\_Task, Set\_CPU, Get\_CPU and Delay\_Until\_And\_Set\_CPU atomically with respect to any of these operations on the same dispatching\_domain, processor or task.

**the new paragraph:**

Any task that belongs to the system dispatching domain can execute on any CPU within that domain, unless the assignment of the task has been specified.

## Annex E: Distributed Systems

### E.2.1 Shared Passive Library Units

**Replace paragraph 7: [8652/0195]**

- it shall not contain a library-level declaration of an access type that designates a class-wide type, task type, or protected type with `entry_declarations`.

**by:**

- it shall not contain a library-level declaration of an access type that designates a class-wide type, nor a type with a part that is of a task type or protected type with `entry_declarations`.

**Replace paragraph 8: [8652/0195]**

Notwithstanding the definition of accessibility given in 3.10.2, the declaration of a library unit P1 is not accessible from within the declarative region of a shared passive library unit P2, unless the shared passive library unit P2 depends semantically on P1.

**by:**

Notwithstanding the definition of accessibility given in 3.10.2, the declaration of a library unit P1 is not accessible from within the declarative region of a shared passive library unit P2, unless the shared passive library unit P2 depends semantically on P1. Furthermore, for the purposes of accessibility checking, when an access type that is declared within a declared-pure package is used as part of a library-level declaration in a shared-passive package, it is as though the access type were declared in the shared-passive package.

### E.2.2 Remote Types Library Units

**Replace paragraph 16: [8652/0196]**

- A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call where the value designates a controlling operand of the call (see E.4, "Remote Subprogram Calls");

**by:**

- A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call to a primitive operation of the designated type where the value designates a controlling operand of the call (see E.4, "Remote Subprogram Calls");

**Replace paragraph 17: [8652/0159; 8652/0197]**

- The `Storage_Pool` attribute is not defined for a remote access-to-class-wide type; the expected type for an `allocator` shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The `Storage_Size` attribute of a remote access-to-class-wide type yields 0; it is not allowed in an `attribute_definition_clause`.

**by:**

- The `Storage_Pool` attribute is not defined for a remote access-to-class-wide type; the expected type for an `allocator` shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The `Storage_Size` attribute of a remote access-to-class-wide type yields 0. The `Storage_Pool` and `Storage_Size` aspects shall not be specified for a remote access-to-class-wide type.

*Erroneous Execution*

Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared in the visible part of a remote types package.



### E.2.3 Remote Call Interface Library Units

**Replace paragraph 19: [8652/0198]**

If aspect `All_Calls_Remote` is True for a given RCI library unit, then the implementation shall route any call to a subprogram of the RCI unit from outside the declarative region of the unit through the Partition Communication Subsystem (PCS); see E.5. Calls to such subprograms from within the declarative region of the unit are defined to be local and shall not go through the PCS.

**by:**

If aspect `All_Calls_Remote` is True for a given RCI library unit, then the implementation shall route any of the following calls through the Partition Communication Subsystem (PCS); see E.5:

- A direct call to a subprogram of the RCI unit from outside the declarative region of the unit;
- An indirect call through a remote access-to-subprogram value that designates a subprogram of the RCI unit;
- A dispatching call with a controlling operand designated by a remote access-to-class-wide value whose tag identifies a type declared in the RCI unit.

## **Annex F: Information Systems**

No changes in this clause.

## **Annex G: Numerics**

No changes in this clause.

## **Annex H: Safety and Security**

No changes in this clause.

## **Annex J: Obsolescent Features**

No changes in this clause.

## **Annex M: Implementation-Defined Characteristics**

No changes in this clause.

## Annex N: Glossary

### N Glossary

**Replace paragraph 21.2: [8652/0124]**

**Invariant.** A invariant is an assertion that is expected to be True for all objects of a given private type when viewed from outside the defining package.

**by:**

**Invariant.** An invariant is an assertion that is expected to be True for all objects of a given private type when viewed from outside the defining package.

**Insert after paragraph 41: [8652/0124]**

**Type.** Each object has a type. A *type* has an associated set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. Types are grouped into *categories*. Most language-defined categories of types are also *classes* of types.

**the new paragraph:**

**Type Invariant.** See Invariant.

## **Annex Q: Language-Defined Entities**

No changes in this clause.