**ISO/IEC JTC1/SC22/WG9 N377**
**2 August 2000**

**DRAFT Submitted for WG9 Email Ballot**

# Programming languages — Ada
**TECHNICAL CORRIGENDUM 1**

Langages de programmation — Ada
RECTIFICATIF TECHNIQUE 1

Technical Corrigendum 1 to International Standard ISO/IEC 8652:1995 was prepared by AXE Consulting under contract from The MITRE Corporation.

# Programming languages — Ada

## TECHNICAL CORRIGENDUM 1

*Langages de programmation — Ada*

*RECTIFICATIF TECHNIQUE 1*

Technical Corrigendum 1 to International Standard ISO/IEC 8652:1995 was prepared by AXE Consulting under contract from The MITRE Corporation.

# Introduction

This corrigendum contains corrections to the Ada 95 standard [ISO/IEC 8652:1995].

The corrigendum is organized by sections corresponding to those in the Ada 95 standard. These sections include wording changes to the Ada 95 standard. Clause and subclause headings are given for each clause that contains a wording change. Other clauses 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 95 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 95 standard - this font for text, this font for syntax, `and this font for Ada source code.`

# Section 1: General

## 1.2 Normative References

**Replace paragraph 8: [8652/0001]**

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

**by:**

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

## Section 2: Lexical Elements

No changes in this section.

# Section 3: Declarations and Types

## 3.3.1 Object Declarations

**Replace paragraph 18: [8652/0002]**

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

**by:**

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

## 3.5.4 Integer Types

**Insert after paragraph 27: [8652/0003]**

For a one's complement machine, the high bound of the base range of a modular type whose modulus is one less than a power of 2 may be equal to the modulus, rather than one less than the modulus. It is implementation defined for which powers of 2, if any, this permission is exercised.

**the new paragraph:**

For a one's complement machine, implementations may support non-binary modulus values greater than System.Max_Nonbinary_Modulus. It is implementation defined which specific values greater than System.Max_Nonbinary_Modulus, if any, are supported.

## 3.5.8 Operations of Floating Point Types

**Replace paragraph 2: [8652/0004]**

S'Digits S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type T is defined to be the largest value of *d* for which ceiling(*d* * log(10) / log(T'Machine_Radix)) + 1 <= T'Model_Mantissa.

**by:**

S'Digits S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type T is defined to be the largest value of *d* for which

ceiling(*d* * log(10) / log(T'Machine_Radix)) + *g* <= T'Model_Mantissa

where *g* is 0 if Machine_Radix is a positive power of 10 and 1 otherwise.

## 3.5.10 Operations of Fixed Point Types

**Replace paragraph 2: [8652/0005]**

S'Small S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. Small may be specified for nonderived fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static.

**by:**

S'Small S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. Small may be specified for nonderived ordinary fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static.

## 3.6 Array Types

**Replace paragraph 22: [8652/0002]**

The elaboration of a discrete_subtype_definition creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

**by:**

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

## 3.6.2 Operations of Array Types

**Replace paragraph 2: [8652/0006]**

The following attributes are defined for a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

**by:**

The following attributes are defined for a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

## 3.7 Discriminants

**Replace paragraph 8: [8652/0007]**

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

**by:**

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

## 3.7.1 Discriminant Constraints

**Replace paragraph 7: [8652/0008]**

A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype.

**by:**

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

## 3.8 Record Types

**Replace paragraph 5: [8652/0009]**

component_item ::= component_declaration | representation_clause

**by:**

component_item ::= component_declaration | aspect_clause

**Replace paragraph 18: [8652/0002]**

Within the definition of a composite type, if a component_definition or discrete_subtype_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint being defined is called a *per-object constraint*. For the elaboration of a component_definition of a component_declaration, if the constraint of the subtype_indication is not a per-object constraint, then the subtype_indication is elaborated. On the other hand, if the constraint is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression.

**by:**

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

When a per-object constraint is elaborated (as part of creating an object), each per-object expression of the constraint is evaluated. For other expressions, the values determined during the elaboration of the component_definition or entry_declaration are used. Any checks associated with the enclosing subtype_indication or discrete_subtype_definition are performed, including the subtype compatibility check (see 3.2.2), and the associated subtype is created.

## 3.9.2 Dispatching Operations of Tagged Types

**Replace paragraph 7: [8652/0010]**

A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively. For a controlling operand that is designated by an actual parameter, the controlling operand is statically or dynamically tagged according to whether the designated type of the actual parameter is specific or class-wide, respectively.

**by:**

A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively. For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form X'Access, where X is of a class-wide type, or is of the form **new** T'(...), where T denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively.

**Replace paragraph 9: [8652/0010]**

If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the expression shall not be of an access-to-class-wide type unless it designates a controlling operand in a call on a dispatching operation.

**by:**

If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the object designated by the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation.

**Replace paragraph 10: [8652/0011]**

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. A dispatching operation shall not be of convention Intrinsic. If a dispatching operation overrides the predefined equals operator, then it shall be of convention Ada (either explicitly or by default — see 6.3.1).

**by:**

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

## 3.10 Access Types

**Replace paragraph 7: [8652/0012]**

There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators; storage pools are described further in 13.11, ‘‘Storage Management’’.

**by:**

There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. All descendants of an access type share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators; storage pools are described further in 13.11, ‘‘Storage Management’’.

**Replace paragraph 14: [8652/0013]**

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

**by:**

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

## 3.10.2 Operations of Access Types

**Replace paragraph 24: [8652/0010]**

X'Access

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type *A*:

**by:**

X'Access

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type *A*, with designated type *D*:

**Replace paragraph 27: [8652/0010]**

- If the designated type of *A* is tagged, then the type of the view shall be covered by the designated type; if *A*'s designated type is not tagged, then the type of the view shall be the same, and either *A*'s designated subtype shall statically match the nominal subtype of the view, or the designated subtype shall be discriminated and unconstrained;

**by:**

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

## 3.11 Declarative Parts

**Replace paragraph 4: [8652/0009]**

```
basic_declarative_item ::=
   basic_declaration | representation_clause | use_clause
```

**by:**

```
basic_declarative_item ::=
   basic_declaration | aspect_clause | use_clause
```

**Replace paragraph 10: [8652/0014]**

- For a call to a (non-protected) subprogram that has an explicit body, a check is made that the subprogram_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

**by:**

- For a call to a (non-protected) subprogram that has an explicit body, a check is made that the body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

## 3.11.1 Completions of Declarations

**Replace paragraph 1: [8652/0014]**

Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma.

**by:**

Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma. A *body* is a body, an entry_body, or a renaming-as-body (see 8.5.4).

# Section 4: Names and Expressions

## 4.1.4 Attributes

**Replace paragraph 12: [8652/0015]**

An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes.

**by:**

An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes unless supplied for compatibility with a previous edition of this International Standard.

## 4.5.2 Relational Operators and Membership Tests

**Insert after paragraph 24: [8652/0016]**

- Otherwise, the result is defined in terms of the primitive equals operator for any matching tagged components, and the predefined equals for any matching untagged components.

**the new paragraph:**

For any composite type, the order in which "=" is called for components is not defined by the language. Furthermore, if the result can be determined before calling "=" on some components, the language does not define whether "=" is called on those components.

**Insert after paragraph 32: [8652/0016]**

A membership test using **not in** gives the complementary result to the corresponding membership test using **in**.

**the new paragraph:**

*Implementation Requirements*

For all nonlimited types declared in language-defined packages, the "=" and "/=" operators of the type shall behave as if they were the predefined equality operators for the purposes of the equality of composite types and generic formal types.

## 4.6 Type Conversions

**Replace paragraph 5: [8652/0017]**

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

**by:**

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

**Replace paragraph 11: [8652/0008]**

- Corresponding index types shall be convertible; and

**by:**

- Corresponding index types shall be convertible;

**Replace paragraph 12: [8652/0008]**

- The component subtypes shall statically match.

**by:**

- The component subtypes shall statically match; and
- In a view conversion, the target type and the operand type shall both or neither have aliased components.

**Replace paragraph 54: [8652/0017]**

- If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the operand type is a descendant of the target type, and has discriminants that were not inherited from the target type;

**by:**

- If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the target subtype is indefinite, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type;

## 4.8 Allocators

**Replace paragraph 3: [8652/0010]**

The expected type for an allocator shall be a single access-to-object type whose designated type covers the type determined by the subtype_mark of the subtype_indication or qualified_expression.

**by:**

The expected type for an allocator shall be a single access-to-object type with designated type *D* such that either *D* covers the type determined by the subtype_mark of the subtype_indication or qualified_expression, or the expected type is anonymous and the determined type is *D*'Class.

**Replace paragraph 10: [8652/0002]**

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

**by:**

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

## Section 5: Statements

No changes in this section.

# Section 6: Subprograms

## 6.3.1 Conformance Rules

**Replace paragraph 2: [8652/0011]**

As explained in B.1, ‘‘Interfacing Pragmas’’, a *convention* can be specified for an entity. For a callable entity or access-to-subprogram type, the convention is called the *calling convention*. The following conventions are defined by the language:

**by:**

As explained in B.1, ‘‘Interfacing Pragmas’’, a *convention* can be specified for an entity. Unless this International Standard states otherwise, the default convention of an entity is Ada. For a callable entity or access-to-subprogram type, the convention is called the *calling convention*. The following calling conventions are defined by the language:

**Insert after paragraph 13: [8652/0011]**

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

**the new paragraph:**

- If not specified above as Intrinsic, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type. The default calling convention for a new dispatching operation of a tagged type is the convention of the type.

**Insert after paragraph 21: [8652/0018]**

- 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

**the new paragraph:**

- each attribute_designator in one must be the same as the corresponding attribute_designator in the other; and

# Section 7: Packages

## 7.3.1 Private Operations

**Replace paragraph 3: [8652/0019]**

For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later within the immediate scope of the composite type additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

**by:**

For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

**Replace paragraph 4: [8652/0019]**

The corresponding rule applies to a type defined by a derived_type_definition, if there is a place within its immediate scope where additional characteristics of its parent type become visible.

**by:**

The corresponding rule applies to a type defined by a derived_type_definition, if there is a place immediately within the declarative region in which the type is declared where additional characteristics of its parent type become visible.

**Replace paragraph 5: [8652/0019]**

For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place within the immediate scope of the array type. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.

**by:**

For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place immediately within the declarative region in which the array type is declared. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.

**Replace paragraph 6: [8652/0019]**

Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, within the immediate scope of the type_declaration, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type, it is possible to dispatch to it.

**by:**

Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the type_declaration occurs, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type it is possible to dispatch to it.

## 7.6 User-Defined Assignment and Finalization

**Replace paragraph 4: [8652/0020]**

```
package Ada.Finalization is
    pragma Preelaborate(Finalization);
```

**by:**

```
package Ada.Finalization is
    pragma Preelaborate(Finalization);
    pragma Remote_Types(Finalization);
```

**Replace paragraph 11: [8652/0021]**

For an extension_aggregate whose ancestor_part is a subtype_mark, Initialize is called on all controlled subcomponents of the ancestor part; if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

**by:**

For an extension_aggregate whose ancestor_part is a subtype_mark, for each controlled subcomponent of the ancestor part, either Initialize is called, or its initial value is assigned, as appropriate; if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

**Insert after paragraph 17: [8652/0022]**

For an assignment_statement, after the name and expression have been evaluated, and any conversion (including constraint checking) has been done, an anonymous object is created, and the value is assigned into it; that is, the assignment operation is applied. (Assignment includes value adjustment.) The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. As explained below, the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in 5.2, ‘‘Assignment Statements’’.

**the new paragraph:**

*Implementation Requirements*

For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement or a return_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

## 7.6.1 Completion and Finalization

**Replace paragraph 13: [8652/0021; 8652/0023]**

The anonymous objects created by function calls and by aggregates are finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, they are finalized before starting the execution of any statement within the compound_statement.

**by:**

If the object_name in an object_renaming_declaration, or the actual parameter for a generic formal **in out** parameter in a generic_instantiation, denotes any part of an anonymous object created by a function call, the anonymous object is not finalized until after it is no longer accessible via any name. Otherwise, an anonymous object created by a function call or by an aggregate is finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, the object is finalized before starting the execution of any statement within the compound_statement.

If a transfer of control or raising of an exception occurs prior to performing a finalization of an anonymous object, the anonymous object is finalized as part of the finalizations due to be performed for the object's innermost enclosing master.

**Replace paragraph 14: [8652/0023]**

It is a bounded error for a call on Finalize or Adjust to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation:

**by:**

It is a bounded error for a call on Finalize or Adjust that occurs as part of object finalization or assignment to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation:

**Replace paragraph 16: [8652/0024]**

- For an Adjust invoked as part of an assignment operation, any other adjustments due to be performed are performed, and then Program_Error is raised.

**by:**

- For an Adjust invoked as part of the initialization of a controlled object, other adjustments due to be performed might or might not be performed, and then Program_Error is raised. During its propagation, finalization might or might not be applied to objects whose Adjust failed. For an Adjust invoked as part of an assignment statement, any other adjustments due to be performed are performed, and then Program_Error is raised.

**Insert after paragraph 17: [8652/0023]**

- For a Finalize invoked as part of a call on an instance of Unchecked_Deallocation, any other finalizations due to be performed are performed, and then Program_Error is raised.

**the new paragraphs:**

- For a Finalize invoked as part of the finalization of the anonymous object created by a function call or aggregate, any other finalizations due to be performed are performed, and then Program_Error is raised.

- For a Finalize invoked due to reaching the end of the execution of a master, any other finalizations associated with the master are performed, and Program_Error is raised immediately after leaving the master.

# Section 8: Visibility Rules

## 8.3 Visibility

**Replace paragraph 9: [8652/0025]**

Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below). A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

**by:**

Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below). The only declarations that are *overridable* are the implicit declarations for predefined operators and inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

**Replace paragraph 10: [8652/0025]**

- An explicit declaration overrides an implicit declaration of a primitive subprogram, regardless of which declaration occurs first;

**by:**

- A declaration that is not overridable overrides one that is overridable, regardless of which declaration occurs first;

**Replace paragraph 26: [8652/0025; 8652/0026]**

An explicit declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the explicit declaration. Similarly, the context_clause for a subunit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

**by:**

A non-overridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the non-overridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the context_clause for a subunit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

## 8.5.1 Object Renaming Declarations

**Replace paragraph 5: [8652/0017]**

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. A slice of an array shall not be renamed if this restriction disallows renaming of the array.

**by:**

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. These rules also apply for a renaming that appears in the body of a generic unit, with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of an untagged generic formal derived type.

## 8.5.4 Subprogram Renaming Declarations

**Replace paragraph 5: [8652/0027; 8652/0028]**

The profile of a renaming-as-body shall be subtype-conformant with that of the renamed callable entity, and shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the subprogram it declares takes its convention from the renamed subprogram; otherwise the convention of the renamed subprogram shall not be Intrinsic.

**by:**

The profile of a renaming-as-body shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the profile shall be mode-conformant with that of the renamed callable entity and the subprogram it declares takes its convention from the renamed subprogram; otherwise, the profile shall be subtype-conformant with that of the renamed callable entity and the convention of the renamed subprogram shall not be Intrinsic. A renaming-as-body is illegal if the declaration occurs before the subprogram whose declaration it completes is frozen, and the renaming renames the subprogram itself, through one or more subprogram renaming declarations, none of whose subprograms has been frozen.

**Replace paragraph 8: [8652/0014; 8652/0027]**

For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called.

**by:**

For a call to a subprogram whose body is given as a renaming-as-body, the execution of the renaming-as-body is equivalent to the execution of a subprogram_body that simply calls the renamed subprogram with its formal parameters as the actual parameters and, if it is a function, returns the value of the call.

For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called.

*Bounded (Run-Time) Errors*

If a subprogram directly or indirectly renames itself, then it is a bounded error to call that subprogram. Possible consequences are that Program_Error or Storage_Error is raised, or that the call results in infinite recursion.

# Section 9: Tasks and Synchronization

## 9.1 Task Units and Task Objects

**Replace paragraph 5: [8652/0009]**

```
task_item ::= entry_declaration | representation_clause
```

**by:**

```
task_item ::= entry_declaration | aspect_clause
```

**Insert after paragraph 9: [8652/0029]**

A task_definition defines a task type and its first subtype. The first list of task_items of a task_definition, together with the known_discriminant_part, if any, is called the visible part of the task unit. The optional list of task_items after the reserved word **private** is called the private part of the task unit.

**the new paragraph:**

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

**Replace paragraph 12: [8652/0009]**

As part of the initialization of a task object, any representation_clauses and any per-object constraints associated with entry_declarations of the corresponding task_definition are elaborated in the given order.

**by:**

As part of the initialization of a task object, any aspect_clauses and any per-object constraints associated with entry_declarations of the corresponding task_definition are elaborated in the given order.

## 9.4 Protected Units and Protected Objects

**Replace paragraph 5: [8652/0009]**

```
protected_operation_declaration ::= subprogram_declaration
        | entry_declaration
        | representation_clause
```

**by:**

```
protected_operation_declaration ::= subprogram_declaration
        | entry_declaration
        | aspect_clause
```

**Replace paragraph 8: [8652/0009]**

```
protected_operation_item ::= subprogram_declaration
        | subprogram_body
        | entry_body
        | representation_clause
```

**by:**

```
protected_operation_item ::= subprogram_declaration
        | subprogram_body
        | entry_body
        | aspect_clause
```

## 9.5.2 Entries and Accept Statements

**Replace paragraph 22: [8652/0002]**

For the elaboration of an entry_declaration for an entry family, if the discrete_subtype_definition contains no per-object expressions (see 3.8), then the discrete_subtype_definition is elaborated. Otherwise, the elaboration of the entry_declaration consists of the evaluation of any expression of the discrete_subtype_definition that is not a per-object expression (or part of one). The elaboration of an entry_declaration for a single entry has no effect.

**by:**

The elaboration of an entry_declaration for an entry family consists of the elaboration of the discrete_subtype_definition, as described in 3.8. The elaboration of an entry_declaration for a single entry has no effect.

## 9.6 Delay Statements, Duration, and Time

**Replace paragraph 26: [8652/0030]**

The exception Time_Error is raised by the function Time_Of if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "-" if the result is not representable in the type Time or Duration, as appropriate. This exception is also raised by the function Year or the procedure Split if the year number of the given date is outside of the range of the subtype Year_Number.

**by:**

The exception Time_Error is raised by the function Time_Of if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "-" if the result is not representable in the type Time or Duration, as appropriate. This exception is also raised by the functions Year, Month, Day, and Seconds and the procedure Split if the year number of the given date is outside of the range of the subtype Year_Number.

## 9.10 Shared Variables

**Insert after paragraph 6: [8652/0031]**

- If A1 is part of the execution of a task, and A2 is the action of waiting for the termination of the task;

**the new paragraph:**

- If A1 is the termination of a task T, and A2 is either the evaluation of the expression T'Terminated or a call to Ada.Task_Identification.Is_Terminated with an actual parameter that identifies T (see C.7.1);

# Section 10: Program Structure and Compilation Issues

## 10.1.4 The Compilation Process

**Replace paragraph 4: [8652/0032]**

If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration for a subprogram or a generic subprogram with the same defining_program_unit_name already exists in the environment (even if the profile of the body is not type conformant with that of the declaration); otherwise the subprogram_body is interpreted as both the declaration and body of a library subprogram.

**by:**

If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration with the same defining_program_unit_name already exists in the environment for a subprogram other than an instance of a generic subprogram or for a generic subprogram (even if the profile of the body is not type conformant with that of the declaration); otherwise the subprogram_body is interpreted as both the declaration and body of a library subprogram.

## 10.1.5 Pragmas and Program Units

**Replace paragraph 5: [8652/0033]**

- Immediately within the declaration of a program unit and before any nested declaration, in which case the argument, if any, shall be a direct_name that denotes the immediately enclosing program unit declaration.

**by:**

- Immediately within the visible part of a program unit and before any nested declaration (but not within a generic formal part), in which case the argument, if any, shall be a direct_name that denotes the immediately enclosing program unit declaration.

**Insert after paragraph 7: [8652/0034]**

Certain program unit pragmas are defined to be *library unit pragmas*. The name, if any, in a library unit pragma shall denote the declaration of a library unit.

**the new paragraphs:**

*Static Semantics*

A library unit pragma that applies to a generic unit does not apply to its instances, unless a specific rule for the pragma specifies the contrary.

*Implementation Advice*

When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance.

## 10.2.1 Elaboration Control

**Replace paragraph 11: [8652/0035]**

If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is *preelaborated*. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition. All compilation units of a preelaborated library unit shall be preelaborable. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

**by:**

If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is *preelaborated*. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

# Section 11: Exceptions

## 11.5 Suppressing Checks

**Replace paragraph 11: [8652/0036]**

Access_Check

When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When passing an actual parameter to a formal access parameter, check that the value of the actual parameter is not **null**.

**by:**

Access_Check

When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When passing an actual parameter to a formal access parameter, check that the value of the actual parameter is not **null**. When evaluating a discriminant_association for an access discriminant, check that the value of the discriminant is not **null**.

# Section 12: Generic Units

## 12.5 Formal Types

**Replace paragraph 8: [8652/0037]**

The formal type also belongs to each class that contains the determined class. The primitive subprograms of the type are as for any type in the determined class. For a formal type other than a formal derived type, these are the predefined operators of the type; they are implicitly declared immediately after the declaration of the formal type. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

**by:**

The formal type also belongs to each class that contains the determined class. The primitive subprograms of the type are as for any type in the determined class. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later in its immediate scope according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

## 12.5.1 Formal Private and Derived Types

**Replace paragraph 21: [8652/0038]**

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type, and are implicitly declared at the earliest place, if any, within the immediate scope of the formal type, where the corresponding primitive subprogram of the ancestor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor, even if this primitive has been overridden for the actual type. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

**by:**

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type, and are implicitly declared at the earliest place, if any, within the immediate scope of the formal type, where the corresponding primitive subprogram of the ancestor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

## 12.7 Formal Packages

**Insert after paragraph 8: [8652/0039]**

- For other kinds of formals, the actuals match if they statically denote the same entity.

**the new paragraph:**

For the purposes of matching, any actual parameter that is the name of a formal object of mode **in** is replaced by the formal object's actual expression (recursively).

## Section 13: Representation Issues

**Replace paragraph 1: [8652/0009]**

This section describes features for querying and controlling aspects of representation and for interfacing to hardware.

**by:**

This section describes features for querying and controlling certain aspects of entities and for interfacing to hardware.

### 13.1 Representation Items

**Replace the title: [8652/0009]**

Representation Items

**by:**

Operational and Representation Items

**Replace paragraph 1: [8652/0009]**

There are three kinds of *representation items*: representation_clauses, component_clauses, and *representation pragmas*. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware). Representation items also specify other specifiable properties of entities. A representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation.

**by:**

Representation and operational items can be used to specify aspects of entities. Two kinds of aspects of entities can be specified: aspects of representation and operational aspects. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. Operational items specify other properties of entities.

There are six kinds of *representation items*: attribute_definition_clauses for representation attributes, enumeration_representation_clauses, record_representation_clauses, at_clauses, component_clauses, and *representation pragmas*. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).

An *operational item* is an attribute_definition_clause for an operational attribute.

An operational item or a representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation.

**Replace paragraph 2: [8652/0009]**

```
representation_clause ::= attribute_definition_clause
      | enumeration_representation_clause
      | record_representation_clause
      | at_clause
```

**by:**

```
aspect_clause ::= attribute_definition_clause
      | enumeration_representation_clause
      | record_representation_clause
      | at_clause
```

**Replace paragraph 4: [8652/0009]**

A representation pragma is allowed only at places where a representation_clause or compilation_unit is allowed.

**by:**

A representation pragma is allowed only at places where an aspect_clause or compilation_unit is allowed.

**Replace paragraph 5: [8652/0009]**

In a representation item, if the local_name is a direct_name, then it shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative_region as the representation item. If the local_name has an attribute_designator, then it shall resolve to denote an implementation-defined component (see 13.5.1) or a class-wide type implicitly declared immediately within the same declarative_region as the representation item. A local_name that is a *library_unit_*name (only permitted in a representation pragma) shall resolve to denote the library_item that immediately precedes (except for other pragmas) the representation pragma.

**by:**

In an operational item or representation item, if the local_name is a direct_name, then it shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative_region as the item. If the local_name has an attribute_designator, then it shall resolve to denote an implementation-defined component (see 13.5.1) or a class-wide type implicitly declared immediately within the same declarative_region as the item. A local_name that is a *library_unit_*name (only permitted in a representation pragma) shall resolve to denote the library_item that immediately precedes (except for other pragmas) the representation pragma.

**Replace paragraph 6: [8652/0009]**

The local_name of a representation_clause or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same declarative_part, package_specification, task_definition, protected_definition, or record_definition as the representation item. If a local_name denotes a local callable entity, it may do so through a local subprogram_renaming_declaration (as a way to resolve ambiguity in the presence of overloading); otherwise, the local_name shall not denote a renaming_declaration.

**by:**

The local_name of an aspect_clause or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same declarative_part, package_specification, task_definition, protected_definition, or record_definition as the representation or operational item. If a local_name denotes a local callable entity, it may do so through a local subprogram_renaming_declaration (as a way to resolve ambiguity in the presence of overloading); otherwise, the local_name shall not denote a renaming_declaration.

**Insert after paragraph 8: [8652/0009]**

A representation item *directly specifies* an *aspect of representation* of the entity denoted by the local_name, except in the case of a type-related representation item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the subtype's type. A representation item that names a subtype is either *subtype-specific* (Size and Alignment clauses) or *type-related* (all others). Subtype-specific aspects may differ for different subtypes of the same type.

**the new paragraph:**

An operational item *directly specifies* an *operational aspect* of the type of the subtype denoted by the local_name. The local_name of an operational item shall denote a first subtype. Operational items are type-related.

**Insert after paragraph 9: [8652/0009]**

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 is given that directly specifies an aspect of an entity, then it is illegal to give another representation item that directly specifies the same aspect of the entity.

**the new paragraph:**

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

**Replace paragraph 11: [8652/0009; 8652/0011]**

Representation aspects of a generic formal parameter are the same as those of the actual. A type-related representation item is not allowed for a descendant of a generic formal untagged type.

**by:**

Operational and representation aspects of a generic formal parameter are the same as those of the actual. Operational and representation aspects of a partial view are the same as those of the full view. A type-related representation item is not allowed for a descendant of a generic formal untagged type.

**Replace paragraph 13: [8652/0009]**

A representation item that is not supported by the implementation is illegal, or raises an exception at run time.

**by:**

A representation or operational item that is not supported by the implementation is illegal, or raises an exception at run time.

**Replace paragraph 15: [8652/0040]**

A derived type inherits each type-related aspect of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific aspect of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited aspect of representation is overridden by a subsequent representation item that specifies the same aspect of the type or subtype.

**by:**

A derived type inherits each type-related aspect of representation of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific aspect of representation of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited aspect of representation is overridden by a subsequent representation item that specifies the same aspect of the type or subtype.

In contrast, whether operational aspects are inherited by a derived type depends on each specific aspect. When operational aspects are inherited by a derived type, aspects that were directly specified before the declaration of the derived type, or (in the case where the parent is derived) that were inherited by the parent type from the grandparent type are inherited. An inherited operational aspect is overridden by a subsequent operational item that specifies the same aspect of the type.

**Insert after paragraph 18: [8652/0040]**

- If an aspect of representation of an entity is not specified, it is chosen by default in an unspecified manner.

**the new paragraph:**

If an operational aspect is *specified* for an entity (meaning that it is either directly specified or inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value for that aspect.

**Replace paragraph 19: [8652/0009]**

For the elaboration of a representation_clause, any evaluable constructs within it are evaluated.

**by:**

For the elaboration of an aspect_clause, any evaluable constructs within it are evaluated.

## 13.3 Representation Attributes

**Replace the title: [8652/0009]**

Representation Attributes

**by:**

Operational and Representation Attributes

**Replace paragraph 1: [8652/0009]**

The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate representation attributes. Some of these attributes are specifiable via an attribute_definition_clause.

**by:**

The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate operational or representation attributes. Some of these attributes are specifiable via an attribute_definition_clause.

**Replace paragraph 5: [8652/0009]**

An attribute_designator is allowed in an attribute_definition_clause only if this International Standard explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an aspect of representation.

**by:**

An attribute_designator is allowed in an attribute_definition_clause only if this International Standard explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an operational aspect or an aspect of representation.

**Replace paragraph 9: [8652/0009]**

The following attributes are defined:

**by:**

The following representation attributes are defined: Address, Alignment, Size, Storage_Size, and Component_Size.

**Replace paragraph 74: [8652/0009]**

For every subtype S of a tagged type T (specific or class-wide), the following attribute is defined:

**by:**

The following operational attribute is defined: External_Tag.

For every subtype S of a tagged type T (specific or class-wide):

**Replace paragraph 75: [8652/0040]**

S'External_Tag

S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 3.9.2 and 13.13.2.

**by:**

S'External_Tag

S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 3.9.2 and 13.13.2. The value of External_Tag is never inherited; the default value is always used unless a new value is directly specified for a type.

## 13.4 Enumeration Representation Clauses

**Replace paragraph 11: [8652/0009]**

NOTES

11 Unchecked_Conversion may be used to query the internal codes used for an enumeration type. The attributes of the type, such as Succ, Pred, and Pos, are unaffected by the representation_clause. For example, Pos always returns the position number, *not* the internal integer code that might have been specified in a representation_clause.

**by:**

NOTES

11 Unchecked_Conversion may be used to query the internal codes used for an enumeration type. The attributes of the type, such as Succ, Pred, and Pos, are unaffected by the enumeration_representation_clause. For example, Pos always returns the position number, *not* the internal integer code that might have been specified in an enumeration_representation_clause.

## 13.11 Storage Management

**Replace paragraph 12: [8652/0009]**

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

**by:**

For every access subtype S, the following representation attributes are defined:

**Replace paragraph 39: [8652/0041]**

```
type Mark_Release_Pool_Type
    (Pool_Size : Storage_Elements.Storage_Count;
     Block_Size : Storage_Elements.Storage_Count)
        is new Root_Storage_Pool with limited private;
```

**by:**

```
type Mark_Release_Pool_Type
    (Pool_Size : Storage_Elements.Storage_Count;
     Block_Size : Storage_Elements.Storage_Count)
        is new Root_Storage_Pool with private;
```

## 13.12 Pragma Restrictions

**Insert after paragraph 8: [8652/0042]**

A pragma Restrictions is a configuration pragma; unless otherwise specified for a particular restriction, a partition shall obey the restriction if a pragma Restrictions applies to any compilation unit included in the partition.

**the new paragraphs:**

For the purpose of checking whether a partition contains constructs that violate any restriction (unless specified otherwise for a particular restriction):

- Generic instances are logically expanded at the point of instantiation;
- If an object of a type is declared or allocated and not explicitly initialized, then all expressions appearing in the definition for the type and any of its ancestors are presumed to be used;
- A default_expression for a formal parameter or a generic formal object is considered to be used if and only if the corresponding actual parameter is not provided in a given call or instantiation.

**Insert after paragraph 9: [8652/0042; 8652/0043]**

An implementation may place limitations on the values of the expression that are supported, and limitations on the supported combinations of restrictions. The consequences of violating such limitations are implementation defined.

**the new paragraphs:**

An implementation is permitted to omit restriction checks for code that is recognized at compile time to be unreachable and for which no code is generated.

Whenever enforcement of a restriction is not required prior to execution, an implementation may nevertheless enforce the restriction prior to execution of a partition to which the restriction applies, provided that every execution of the partition would violate the restriction.

## 13.13.1 The Package Streams

**In paragraph 4 replace: [8652/0044]**

```
type Stream_Element_Array is
    array(Stream_Element_Offset range <>) of Stream_Element;
```

**by:**

```
type Stream_Element_Array is
    array(Stream_Element_Offset range <>) of aliased Stream_Element;
```

**Insert after paragraph 9: [8652/0044]**

The Write operation appends Item to the specified stream.

**the new paragraph:**

*Implementation Permissions*

If Stream_Element'Size is not a multiple of System.Storage_Unit, then the components of Stream_Element_Array need not be aliased.

## 13.13.2 Stream-Oriented Attributes

**Replace paragraph 1: [8652/0009]**

The Write, Read, Output, and Input attributes convert values to a stream of elements and reconstruct values from a stream.

**by:**

The operational attributes Write, Read, Output, and Input convert values to a stream of elements and reconstruct values from a stream.

**Replace paragraph 9: [8652/0040]**

For elementary types, the representation in terms of stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in a canonical order. The canonical order of components is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included.

**by:**

For untagged derived types, the Write and Read attributes of the parent type are inherited as specified in 13.1; otherwise, the default implementations of these attributes are used. The default implementations of Write and Read attributes execute as follows:

For elementary types, the representation in terms of stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order, which is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of any ancestor type of *T* has been directly specified and the attribute of any ancestor type of the type of any of the extension components which are of a limited type has not been specified, the attribute of *T* shall be directly specified.

**Replace paragraph 25: [8652/0040]**

Unless overridden by an attribute_definition_clause, these subprograms execute as follows:

**by:**

For untagged derived types, the Output and Input attributes of the parent type are inherited as specified in 13.1; otherwise, the default implementations of these attributes are used. The default implementations of Output and Input attributes execute as follows:

**Insert after paragraph 35: [8652/0045]**

In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or whose component_declaration includes a default_expression, a check is made that the value returned by Read for the component belongs to its subtype. Constraint_Error is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by Read for the component is not a value of its subtype, Constraint_Error is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see 13.9.1).

**the new paragraph:**

In the default implementation of Read and Input for a type, End_Error is raised if the end of the stream is reached before the reading of a value of the type is completed.

**Replace paragraph 36: [8652/0040]**

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. All nonlimited types have default implementations for these operations. An attribute_reference for one of these attributes is illegal if the type is limited, unless the attribute has been specified by an attribute_definition_clause. For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

**by:**

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. All nonlimited types have default implementations for these operations. An attribute_reference for one of these attributes is illegal if the type is limited, unless the attribute has been specified by an attribute_definition_clause or (for a type extension) the attribute has been specified for an ancestor type. For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

*Implementation Requirements*

For every subtype S of a language-defined nonlimited specific type *T*, the output generated by S'Output or S'Write shall be readable by S'Input or S'Read, respectively. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex.

## 13.14 Freezing Rules

**Replace paragraph 3: [8652/0014]**

The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity declared within it, except for incomplete types. A noninstance body causes freezing of each entity declared before it within the same declarative_part.

**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 declared within it, except for incomplete types. A noninstance body other than a renaming-as-body causes freezing of each entity declared before it within the same declarative_part.

**Replace paragraph 4: [8652/0046]**

A construct that (explicitly or implicitly) references an entity can cause the *freezing* of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, expression, or range within the construct causes freezing:

**by:**

A construct that (explicitly or implicitly) references an entity can cause the *freezing* of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, expression, implicit_dereference, or range within the construct causes freezing:

**Replace paragraph 8: [8652/0046]**

A static expression causes freezing where it occurs. A nonstatic expression causes freezing where it occurs, unless the expression is part of a default_expression, a default_name, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct.

**by:**

A static expression causes freezing where it occurs. An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct.

An implicit call freezes the same entities that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions.

If an expression is implicitly converted to a type or subtype *T*, then at the place where the expression causes freezing, *T* is frozen.

**Insert after paragraph 11: [8652/0046]**

- At the place where a name causes freezing, the entity denoted by the name is frozen, unless the name is a prefix of an expanded name; at the place where an object name causes freezing, the nominal subtype associated with the name is frozen.

**the new paragraph:**

- At the place where an implicit_dereference causes freezing, the nominal subtype associated with the implicit_dereference is frozen.

**Replace paragraph 19: [8652/0009]**

A representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.1).

**by:**

An operational item or a representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.1).

# Annex A: Predefined Language Environment

**In paragraph 2 replace: [8652/0047]**

Finalization — 7.6

Interrupts — C.3.2

**by:**

Finalization — 7.6

Float_Text_IO — A.10.9

Float_Wide_Text_IO — A.11

Integer_Text_IO — A.10.8

Integer_Wide_Text_IO — A.11

Interrupts — C.3.2

## A.1 The Package Standard

**Replace paragraph 7: [8652/0028]**

```
-- function "="   (Left, Right : Boolean) return Boolean;
-- function "/="  (Left, Right : Boolean) return Boolean;
-- function "<"   (Left, Right : Boolean) return Boolean;
-- function "<="  (Left, Right : Boolean) return Boolean;
-- function ">"   (Left, Right : Boolean) return Boolean;
-- function ">="  (Left, Right : Boolean) return Boolean;
```

**by:**

```
-- function "="   (Left, Right : Boolean'Base) return Boolean;
-- function "/="  (Left, Right : Boolean'Base) return Boolean;
-- function "<"   (Left, Right : Boolean'Base) return Boolean;
-- function "<="  (Left, Right : Boolean'Base) return Boolean;
-- function ">"   (Left, Right : Boolean'Base) return Boolean;
-- function ">="  (Left, Right : Boolean'Base) return Boolean;
```

**Replace paragraph 9: [8652/0028]**

```
-- function "and" (Left, Right : Boolean) return Boolean;
-- function "or"  (Left, Right : Boolean) return Boolean;
-- function "xor" (Left, Right : Boolean) return Boolean;
```

**by:**

```
-- function "and" (Left, Right : Boolean'Base) return Boolean'Base;
-- function "or"  (Left, Right : Boolean'Base) return Boolean'Base;
-- function "xor" (Left, Right : Boolean'Base) return Boolean'Base;
```

**Replace paragraph 10: [8652/0028]**

```
-- function "not" (Right : Boolean) return Boolean;
```

**by:**

```
-- function "not" (Right : Boolean'Base) return Boolean'Base;
```

## A.4.2 The Package Strings.Maps

**Replace paragraph 63: [8652/0048]**

To_Range returns the Character_Sequence value R, with lower bound 1 and upper bound Map'Length, such that if D = To_Domain(Map) then D(I) maps to R(I) for each I in D'Range.

**by:**

To_Range returns the Character_Sequence value R, such that if D = To_Domain (Map), then R has the same bounds as D, and D(I) maps to R(I) for each I in D'Range.

## A.4.3 Fixed-Length String Handling

**Replace paragraph 68: [8652/0049]**

Find_Token returns in First and Last the indices of the beginning and end of the first slice of Source all of whose elements satisfy the Test condition, and such that the elements (if any) immediately before and after the slice do not satisfy the Test condition. If no such slice exists, then the value returned for Last is zero, and the value returned for First is Source'First.

**by:**

Find_Token returns in First and Last the indices of the beginning and end of the first slice of Source all of whose elements satisfy the Test condition, and such that the elements (if any) immediately before and after the slice do not satisfy the Test condition. If no such slice exists, then the value returned for Last is zero, and the value returned for First is Source'First; however, if Source'First is not in Positive then Constraint_Error is raised.

**Replace paragraph 74: [8652/0049]**

If Low > Source'Last+1, or High < Source'First-1, then Index_Error is propagated. Otherwise, if High >= Low then the returned string comprises Source(Source'First..Low-1) & By & Source(High+1..Source'Last), and if High < Low then the returned string is Insert(Source, Before=>Low, New_Item=>By).

**by:**

If Low > Source'Last+1, or High < Source'First-1, then Index_Error is propagated. Otherwise:

- If High >= Low, then the returned string comprises Source(Source'First..Low-1) & By & Source(High+1..Source'Last), but with lower bound 1.
- If High < Low, then the returned string is Insert(Source, Before=>Low, New_Item=>By).

**Replace paragraph 86: [8652/0049]**

If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""), otherwise it is Source.

**by:**

If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""), otherwise it is Source with lower bound 1.

**Replace paragraph 106: [8652/0049]**

These functions replicate a character or string a specified number of times. The first function returns a string whose length is Left and each of whose elements is Right. The second function returns a string whose length is Left*Right'Length and whose value is the null string if Left = 0 and is (Left-1)*Right & Right otherwise.

**by:**

These functions replicate a character or string a specified number of times. The first function returns a string whose length is Left and each of whose elements is Right. The second function returns a string whose length is Left*Right'Length and whose value is the null string if Left = 0 and otherwise is (Left-1)*Right & Right with lower bound 1.

## A.4.4 Bounded-Length String Handling

**Replace paragraph 101: [8652/0049]**

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1.

**by:**

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source).

**Replace paragraph 105: [8652/0049]**

Each of the transformation subprograms (Replace_Slice, Insert, Overwrite, Delete), selector subprograms (Trim, Head, Tail), and constructor functions ("*") has an effect based on its corresponding subprogram in Strings.Fixed, and Replicate is based on Fixed."*". For each of these subprograms, the corresponding fixed-length string subprogram is applied to the string represented by the Bounded_String parameter. To_Bounded_String is applied the result string, with Drop (or Error in the case of Generic_Bounded_Length."*") determining the effect when the string length exceeds Max_Length.

**by:**

Each of the transformation subprograms (Replace_Slice, Insert, Overwrite, Delete), selector subprograms (Trim, Head, Tail), and constructor functions ("*") has an effect based on its corresponding subprogram in Strings.Fixed, and Replicate is based on Fixed."*". In the case of a function, the corresponding fixed-length string function is applied to the string represented by the Bounded_String parameter. To_Bounded_String is applied to the result string, with Drop (or Error in the case of Generic_Bounded_Length."*") determining the effect when the string length exceeds Max_Length. In the case of a procedure, the corresponding function in Strings.Bounded.Generic_Bounded_Length is applied, with the result assigned into the Source parameter.

## A.5.1 Elementary Functions

**Replace paragraph 9: [8652/0020]**

The library package Numerics.Elementary_Functions defines the same subprograms as Numerics.Generic_Elementary_Functions, except that the predefined type Float is systematically substituted for Float_Type'Base throughout. Nongeneric equivalents of Numerics.Generic_Elementary_Functions for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Elementary_Functions, Numerics.Long_Elementary_Functions, etc.

**by:**

The library package Numerics.Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Elementary_Functions, except that the predefined type Float is systematically substituted for Float_Type'Base throughout. Nongeneric equivalents of Numerics.Generic_Elementary_Functions for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Elementary_Functions, Numerics.Long_Elementary_Functions, etc.

## A.5.2 Random Number Generation

**Replace paragraph 40: [8652/0050]**

Invoking Value with a string that is not the image of any generator state raises Constraint_Error.

**by:**

*Bounded (Run-Time) Errors*

It is a bounded error to invoke Value with a string that is not the image of any generator state. If the error is detected, Constraint_Error or Program_Error is raised. Otherwise, a call to Reset with the resulting state will produce a generator such that calls to Random with this generator will produce a sequence of values of the appropriate subtype, but which might not be random in character. That is, the sequence of values might not fulfill the implementation requirements of this subclause.

## A.10.1 The Package Text_IO

**Replace paragraph 21: [8652/0051]**

*-- Buffer control*

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

**by:**

-- *Buffer control*

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

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

**Replace paragraph 12: [8652/0052]**

Returns the standard error file (see A.10), or an access value designating the standard output file, respectively.

**by:**

Returns the standard error file (see A.10), or an access value designating the standard error file, respectively.

**Replace paragraph 20: [8652/0051]**

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

**by:**

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

**Replace paragraph 22: [8652/0053]**

The execution of a program is erroneous if it attempts to use a current default input, default output, or default error file that no longer exists.

**by:**

The execution of a program is erroneous if it invokes an operation on a current default input, default output, or default error file, and if the corresponding file object is closed or no longer exists.

**Delete paragraph 23: [8652/0053]**

If the Close operation is applied to a file object that is also serving as the default input, default output, or default error file, then subsequent operations on such a default file are erroneous.

## A.10.10 Input-Output for Enumeration Types

**Replace paragraph 17: [8652/0054]**

Although the specification of the generic package Enumeration_IO would allow instantiation for an float type, this is not the intended purpose of this generic package, and the effect of such instantiations is not defined by the language.

**by:**

Although the specification of the generic package Enumeration_IO would allow instantiation for an integer type, this is not the intended purpose of this generic package, and the effect of such instantiations is not defined by the language.

## A.12.1 The Package Streams.Stream_IO

**Insert before paragraph 2: [8652/0055]**

The library package Streams.Stream_IO has the following declaration:

**the new paragraph:**

The elements of a stream file are stream elements. If positioning is supported for the specified external file, a current index and current size are maintained for the file as described in A.8. If positioning is not supported, a current index is not maintained, and the current size is implementation defined.

**Replace paragraph 25: [8652/0051]**

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

**by:**

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

**Insert after paragraph 28: [8652/0055]**

The subprograms Create, Open, Close, Delete, Reset, Mode, Name, Form, Is_Open, and End_of_File have the same effect as the corresponding subprograms in Sequential_IO (see A.8.2).

**the new paragraphs:**

The Set_Mode procedure changes the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

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.

**Replace paragraph 29: [8652/0056]**

The Stream function returns a Stream_Access result from a File_Type object, thus allowing the stream-oriented attributes Read, Write, Input, and Output to be used on the same file for multiple types.

**by:**

The Stream function returns a Stream_Access result from a File_Type object, thus allowing the stream-oriented attributes Read, Write, Input, and Output to be used on the same file for multiple types. Stream propagates Status_Error if File is not open.

**Insert after paragraph 30: [8652/0055]**

The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index.

**the new paragraph:**

The Size function returns the current size of the file.

**Replace paragraph 31: [8652/0055]**

The Index function returns the current file index, as a count (in stream elements) from the beginning of the file. The position of the first element in the file is 1.

**by:**

The Index function returns the current index.

**Insert after paragraph 32: [8652/0055]**

The Set_Index procedure sets the current index to the specified value.

**the new paragraphs:**

If positioning is supported for the external file, the current index is maintained as follows:

- For Open and Create, if the Mode parameter is Append_File, the current index is set to the current size of the file; otherwise, the current index is set to one.
- For Reset, if the Mode parameter is Append_File, or no Mode parameter is given and the current mode is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.
- For Set_Mode, if the new mode is Append_File, the current index is set to current size plus one; otherwise, the current index is unchanged.
- For Read and Write without a Positive_Count parameter, the current index is incremented by the number of stream elements read or written.

- For Read and Write with a Positive_Count parameter, the value of the current index is set to the value of the Positive_Count parameter plus the number of stream elements read or written.

**Delete paragraph 34: [8652/0055]**

The Size function returns the current size of the file, in stream elements.

**Delete paragraph 35: [8652/0055]**

The Set_Mode procedure changes the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

**Replace paragraph 36: [8652/0055; 8652/0056]**

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.

**by:**

*Erroneous Execution*

If the File_Type object passed to the Stream function is later closed or finalized, and the stream-oriented attributes are subsequently called (explicitly or implicitly) on the Stream_Access value returned by Stream, execution is erroneous. This rule applies even if the File_Type object was opened again after it had been closed.

## A.14 File Sharing

**Delete paragraph 3: [8652/0057]**

- Standard_Input and Standard_Output are associated with distinct external files, so operations on one of these files cannot affect operations on the other file. In particular, reading from Standard_Input does not affect the current page, line, and column numbers for Standard_Output, nor does writing to Standard_Output affect the current page, line, and column numbers for Standard_Input.

# Annex B: Interface to Other Languages

## B.1 Interfacing Pragmas

**Insert after paragraph 9: [8652/0058]**

A pragma Linker_Options is allowed only at the place of a declarative_item.

**the new paragraph:**

For pragmas Import and Export, the argument for Link_Name shall not be given without the pragma_argument_identifier unless the argument for External_Name is given.

## B.3 Interfacing with C

**Replace paragraph 1: [8652/0059]**

The facilities relevant to interfacing with the C language are the package Interfaces.C and its children; and support for the Import, Export, and Convention pragmas with *convention_*identifier C.

**by:**

The facilities relevant to interfacing with the C language are the package Interfaces.C and its children; support for the Import, Export, and Convention pragmas with *convention_*identifier C; and support for the Convention pragma with *convention_*identifier C_Pass_By_Copy.

**Replace paragraph 20: [8652/0060]**

```
nul : constant char := char'First;
```

**by:**

```
nul : constant char := implementation-defined;
```

**Replace paragraph 30: [8652/0060]**

```
type wchar_t is implementation-defined;
```

**by:**

```
type wchar_t is <implementation-defined discrete type>;
```

**Replace paragraph 31: [8652/0060]**

```
wide_nul : constant wchar_t := wchar_t'First;
```

**by:**

```
wide_nul : constant wchar_t := implementation-defined;
```

**Insert after paragraph 60: [8652/0059]**

The To_C and To_Ada subprograms that convert between Wide_String and wchar_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that wide_nul is used instead of nul.

**the new paragraphs:**

A Convention pragma with *convention_*identifier C_Pass_By_Copy shall only be applied to a type.

The eligibility rules in B.1 do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible.

If a type is C_Pass_By_Copy-compatible then it is also C-compatible.

**Replace paragraph 61: [8652/0059]**

An implementation shall support pragma Convention with a C *convention_*identifier for a C-eligible type (see B.1)

**by:**

An implementation shall support pragma Convention with a C *convention*_identifier for a C-eligible type (see B.1). An implementation shall support pragma Convention with a C_Pass_By_Copy *convention*_identifier for a C_Pass_By_Copy-eligible type.

**Insert before paragraph 63: [8652/0060]**

An implementation should support the following interface correspondences between Ada and C.

**the new paragraph:**

The constants nul and wide_nul should have a representation of zero.

**Insert after paragraph 68: [8652/0059]**

- An Ada **access** T parameter, or an Ada **out** or **in out** parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary **out** or **in out** parameter, a pointer to a temporary copy is used to preserve by-copy semantics.

**the new paragraph:**

- An Ada parameter of a C_Pass_By_Copy-compatible (record) type T, of mode **in**, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

**Replace paragraph 69: [8652/0059]**

- An Ada parameter of a record type T, of any mode, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

**by:**

- An Ada parameter of a record type T, of any mode, other than an **in** parameter of a C_Pass_By_Copy-compatible type, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

## B.3.1 The Package Interfaces.C.Strings

**Replace paragraph 24: [8652/0061]**

If Item is **null**, then To_Chars_Ptr returns Null_Ptr. Otherwise, if Nul_Check is True and Item.**all** does not contain nul, then the function propagates Terminator_Error; if Nul_Check is True and Item.**all** does contain nul, To_Chars_Ptr performs a pointer conversion with no allocation of memory.

**by:**

If Item is **null**, then To_Chars_Ptr returns Null_Ptr. If Item is not **null**, Nul_Check is True, and Item.**all** does not contain nul, then the function propagates Terminator_Error; otherwise To_Chars_Ptr performs a pointer conversion without allocation of memory.

**Replace paragraph 36: [8652/0062]**

If Item = Null_Ptr then Value(Item) propagates Dereference_Error. Otherwise Value returns the shorter of two arrays: the first Length chars pointed to by Item, and Value(Item). The lower bound of the result is 0.

**by:**

If Item = Null_Ptr, then Value propagates Dereference_Error. Otherwise, Value returns the shorter of two arrays, either the first Length chars pointed to by Item, or Value(Item). The lower bound of the result is 0. If Length is 0, then Value propagates Constraint_Error.

**Replace paragraph 40: [8652/0063]**

Equivalent to To_Ada(Value(Item, Length), Trim_Nul=>True).

**by:**

Equivalent to To_Ada(Value(Item, Length) & nul, Trim_Nul => True).

**Replace paragraph 44: [8652/0064]**

This procedure updates the value pointed to by Item, starting at position Offset, using Chars as the data to be copied into the array. Overwriting the nul terminator, and skipping with the Offset past the nul terminator, are both prevented if Check is True, as follows:

**by:**

If Item = Null_Ptr, then Update propagates Dereference_Error. Otherwise, this procedure updates the value pointed to by Item, starting at position Offset, using Chars as the data to be copied into the array. Overwriting the nul terminator, and skipping with the Offset past the nul terminator, are both prevented if Check is True, as follows:

## B.3.2 The Generic Package Interfaces.C.Pointers

**Replace paragraph 49: [8652/0065]**

```
         loop
            Element             := Source_Temp_Ptr.all;
            Target_Temp_Ptr.all := Element;
            exit when Element = C.nul;
            Char_Ptrs.Increment(Target_Temp_Ptr);
            Char_Ptrs.Increment(Source_Temp_Ptr);
         end loop;
      end Strcpy;
   begin
      ...
   end Test_Pointers;
```

**by:**

```
         loop
            Element             := Source_Temp_Ptr.all;
            Target_Temp_Ptr.all := Element;
            exit when C."="(Element, C.nul);
            Char_Ptrs.Increment(Target_Temp_Ptr);
            Char_Ptrs.Increment(Source_Temp_Ptr);
         end loop;
      end Strcpy;
   begin
      ...
   end Test_Pointers;
```

## B.4 Interfacing with COBOL

**Replace paragraph 63: [8652/0066]**

- Format=Unsigned: if Item comprises zero or more leading space characters followed by one or more decimal digit characters then Valid returns True, else it returns False.

**by:**

- Format=Unsigned: if Item comprises one or more decimal digit characters then Valid returns True, else it returns False.

**Replace paragraph 64: [8652/0066]**

- Format=Leading_Separate: if Item comprises zero or more leading space characters, followed by a single occurrence of the plus or minus sign character, and then one or more decimal digit characters, then Valid returns True, else it returns False.

**by:**

- Format=Leading_Separate: if Item comprises a single occurrence of the plus or minus sign character, and then one or more decimal digit characters, then Valid returns True, else it returns False.

**Replace paragraph 65: [8652/0066]**

- Format=Trailing_Separate: if Item comprises zero or more leading space characters, followed by one or more decimal digit characters and finally a plus or minus sign character, then Valid returns True, else it returns False.

**by:**

- Format=Trailing_Separate: if Item comprises one or more decimal digit characters followed by a plus or minus sign character, then Valid returns True, else it returns False.

**Replace paragraph 71: [8652/0067]**

This function returns the Numeric value for Item, represented in accordance with Format. Conversion_Error is propagated if Num is negative and Format is Unsigned.

**by:**

This function returns the Numeric value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Item is negative and Format is Unsigned.

**Replace paragraph 79: [8652/0067]**

This function returns the Packed_Decimal value for Item, represented in accordance with Format. Conversion_Error is propagated if Num is negative and Format is Packed_Unsigned.

**by:**

This function returns the Packed_Decimal value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Item is negative and Format is Packed_Unsigned.

**Replace paragraph 87: [8652/0067]**

This function returns the Byte_Array value for Item, represented in accordance with Format.

**by:**

This function returns the Byte_Array value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1.

# Annex C: Systems Programming

## C.3.1 Protected Procedure Handlers

**Replace paragraph 12: [8652/0068]**

When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. Otherwise, that is, if an Attach_Handler pragma was used, the previous handler is restored.

**by:**

When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. If an Attach_Handler pragma was used and the most recently attached handler for the same interrupt is the same as the one that was attached at the time the protected object was initialized, the previous handler is restored.

**Insert after paragraph 14: [8652/0068]**

If the Ceiling_Locking policy (see D.3) is in effect and an interrupt is delivered to a handler, and the interrupt hardware priority is higher than the ceiling priority of the corresponding protected object, the execution of the program is erroneous.

**the new paragraph:**

If the handlers for a given interrupt attached via pragma Attach_Handler are not attached and detached in a stack-like (LIFO) order, program execution is erroneous. In particular, when a protected object is finalized, the execution is erroneous if any of the procedures of the protected object are attached to interrupts via pragma Attach_Handler and the most recently attached handler for the same interrupt is not the same as the one that was attached at the time the protected object was initialized.

## C.3.2 The Package Interrupts

**Replace paragraph 16: [8652/0069]**

The Current_Handler function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, Current_Handler returns a value that designates the default treatment; calling Attach_Handler or Exchange_Handler with this value restores the default treatment.

**by:**

The Current_Handler function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, Current_Handler returns **null**.

**Replace paragraph 18: [8652/0069]**

The Exchange_Handler procedure operates in the same manner as Attach_Handler with the addition that the value returned in Old_Handler designates the previous treatment for the specified interrupt.

**by:**

The Exchange_Handler procedure operates in the same manner as Attach_Handler with the addition that the value returned in Old_Handler designates the previous treatment for the specified interrupt. If the previous treatment is not a user-defined handler, **null** is returned.

## C.7.1 The Package Task_Identification

**In paragraph 3 replace: [8652/0070]**

```
procedure Abort_Task   (T : in out Task_Id);
```

**by:**

```
procedure Abort_Task   (T : in Task_Id);
```

## C.7.2 The Package Task_Attributes

**Insert after paragraph 13: [8652/0071]**

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

**the new paragraph:**

*Bounded (Run-Time) Errors*

If the package Ada.Task_Attributes is instantiated with a controlled type and the controlled type has user-defined Adjust or Finalize operations that in turn access task attributes by any of the above operations, then a call of Set_Value of the instantiated package constitutes a bounded error. The call may perform as expected or may result in forever blocking the calling task and subsequently some or all tasks of the partition.

**Insert after paragraph 15: [8652/0071]**

If a value of Task_ID is passed as a parameter to any of the operations declared in this package and the corresponding task object no longer exists, the execution of the program is erroneous.

**the new paragraph:**

Accesses to task attributes via a value of type Attribute_Handle are erroneous if executed concurrently with each other or with calls of any of the operations declared in package Task_Attributes.

**Replace paragraph 16: [8652/0071]**

The implementation shall perform each of the above operations for a given attribute of a given task atomically with respect to any other of the above operations for the same attribute of the same task.

**by:**

For a given attribute of a given task, the implementation shall perform the operations declared in this package atomically with respect to any of these operations of the same attribute of the same task. The granularity of any locking mechanism necessary to achieve such atomicity is implementation defined.

# Annex D: Real-Time Systems

## D.1 Task Priorities

**Replace paragraph 21: [8652/0072]**

- During activation, a task being activated inherits the active priority of the its activator (see 9.2).

**by:**

- During activation, a task being activated inherits the active priority that its activator (see 9.2) had at the time the activation was initiated.

**Replace paragraph 22: [8652/0072]**

- During rendezvous, the task accepting the entry call inherits the active priority of the caller (see 9.5.3).

**by:**

- During rendezvous, the task accepting the entry call inherits the priority of the entry call (see 9.5.3 and D.4).

## D.3 Priority Ceiling Locking

**In paragraph 6 replace: [8652/0073]**

If no Locking_Policy pragma appears in any of the program units comprising a partition, the locking policy for that partition, as well as the effect of specifying either a Priority or Interrupt_Priority pragma for a protected object, are implementation defined.

**by:**

If no Locking_Policy pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the effect of specifying either a Priority or Interrupt_Priority pragma for a protected object, are implementation defined.

## D.4 Entry Queuing Policies

**Replace paragraph 1: [8652/0074]**

This clause specifies a mechanism for a user to choose an entry *queuing policy*. It also defines one such policy. Other policies are implementation defined.

**by:**

This clause specifies a mechanism for a user to choose an entry *queuing policy*. It also defines two such policies. Other policies are implementation defined.

**Replace paragraph 10: [8652/0075]**

- After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set.

**by:**

- After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set while the task is blocked on an entry call.

## D.7 Tasking Restrictions

**Replace paragraph 4: [8652/0042]**

No_Nested_Finalization
Objects with controlled parts and access types that designate such objects shall be declared only at library level.

**by:**

No_Nested_Finalization
: Objects with controlled, protected, or task parts, and access types that designate such objects, shall be declared only at library level.

**Delete paragraph 15: [8652/0076]**

If the following restrictions are violated, the behavior is implementation defined. If an implementation chooses to detect such a violation, Storage_Error should be raised.

**Replace paragraph 17: [8652/0076]**

Max_Storage_At_Blocking
: Specifies the maximum portion (in storage elements) of a task's Storage_Size that can be retained by a blocked task.

**by:**

Max_Storage_At_Blocking
: Specifies the maximum portion (in storage elements) of a task's Storage_Size that can be retained by a blocked task. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

**Replace paragraph 18: [8652/0076]**

Max_Asynchronous_Select_Nesting
: Specifies the maximum dynamic nesting level of asynchronous_selects. A value of zero prevents the use of any asynchronous_select.

**by:**

Max_Asynchronous_Select_Nesting
: Specifies the maximum dynamic nesting of asynchronous_selects. A value of zero prevents the use of any asynchronous_select and, if a program contains an asynchronous_select, it is illegal. If an implementation chooses to detect a violation of this restriction for values other than zero, Storage_Error should be raised; otherwise, the behavior is implementation defined.

**Replace paragraph 19: [8652/0076]**

Max_Tasks
: Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task.

**by:**

Max_Tasks
: Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task. A value of zero prevents any task creation and, if a program contains a task creation, it is illegal. If an implementation chooses to detect a violation of this restriction for values other than zero, Storage_Error should be raised; otherwise, the behavior is implementation defined.

## D.11 Asynchronous Task Control

**Replace paragraph 18: [8652/0077]**

- If a task becomes held while waiting in a selective_accept, and a entry call is issued to one of the open entries, the corresponding accept body executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another Continue.

**by:**

- If a task becomes held while waiting in a selective_accept, and an entry call is issued to one of the open entries, the corresponding accept_alternative executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another Continue.

# Annex E: Distributed Systems

## E.2 Categorization of Library Units

**Replace paragraph 4: [8652/0078]**

A library package or generic library package is called a *shared passive* library unit if a Shared_Passive pragma applies to it. A library package or generic library package is called a *remote types* library unit if a Remote_Types pragma applies to it. A library package or generic library package is called a *remote call interface* if a Remote_Call_Interface pragma applies to it. A *normal library unit* is one to which no categorization pragma applies.

**by:**

A library package or generic library package is called a *shared passive* library unit if a Shared_Passive pragma applies to it. A library package or generic library package is called a *remote types* library unit if a Remote_Types pragma applies to it. A library unit is called a *remote call interface* if a Remote_Call_Interface pragma applies to it. A *normal library unit* is one to which no categorization pragma applies.

**Delete paragraph 13: [8652/0079]**

For a given library-level type declared in a preelaborated library unit or in the declaration of a remote types or remote call interface library unit, the implementation shall choose the same representation for the type upon each elaboration of the type's declaration for different partitions of the same program.

## E.2.1 Shared Passive Library Units

**Replace paragraph 7: [8652/0080]**

- 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; if the shared passive library unit is generic, it shall not contain a declaration for such an access type unless the declaration is nested within a body other than a package_body.

**by:**

- 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.

## E.2.2 Remote Types Library Units

**Replace paragraph 9: [8652/0081; 8652/0082]**

An access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be either an access-to-subprogram type or a general access type that designates a class-wide limited private type.

**by:**

An access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be:

- An access-to-subprogram type, or
- A general access type that designates a class-wide limited private type or a class-wide private type extension all of whose ancestors are either private type extensions or limited private types.

A type that is derived from a remote access type is also a remote access type.

**Replace paragraph 14: [8652/0083]**

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; the types of all the non-controlling formal parameters shall have Read and Write attributes.

**by:**

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall have either a nonlimited type or a type with Read and Write attributes specified via an attribute_definition_clause;

## E.2.3 Remote Call Interface Library Units

**Replace paragraph 7: [8652/0078]**

A *remote call interface (RCI)* is a library unit to which the pragma Remote_Call_Interface applies. A subprogram declared in the visible part of such a library unit is called a *remote subprogram*.

**by:**

A *remote call interface (RCI)* is a library unit to which the pragma Remote_Call_Interface applies. A subprogram declared in the visible part of such a library unit, or declared by such a library unit, is called a *remote subprogram*.

**Replace paragraph 9: [8652/0078]**

In addition, the following restrictions apply to the visible part of an RCI library unit:

**by:**

In addition, the following restrictions apply to an RCI library unit:

**Replace paragraph 10: [8652/0078]**

- it shall not contain the declaration of a variable;

**by:**

- its visible part shall not contain the declaration of a variable;

**Replace paragraph 11: [8652/0078]**

- it shall not contain the declaration of a limited type;

**by:**

- its visible part shall not contain the declaration of a limited type;

**Replace paragraph 12: [8652/0078]**

- it shall not contain a nested generic_declaration;

**by:**

- its visible part shall not contain a nested generic_declaration;

**Replace paragraph 13: [8652/0078]**

- it shall not contain the declaration of a subprogram to which a pragma Inline applies;

**by:**

- it shall not be, nor shall its visible part contain, the declaration of a subprogram to which a pragma Inline applies;

**Replace paragraph 14: [8652/0078]**

- it shall not contain a subprogram (or access-to-subprogram) declaration whose profile has an access parameter, or a formal parameter of a limited type unless that limited type has user-specified Read and Write attributes;

**by:**

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter, or a formal parameter of a limited type unless that limited type has user-specified Read and Write attributes;

**Replace paragraph 19: [8652/0078]**

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

**by:**

If a pragma All_Calls_Remote applies to 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.

## E.3 Consistency of a Distributed System

**Replace paragraph 5: [8652/0084]**

The *version* of a compilation unit changes whenever the version changes for any compilation unit on which it depends semantically. The version also changes whenever the compilation unit itself changes in a semantically significant way. It is implementation defined whether there are other events (such as recompilation) that result in the version of a compilation unit changing.

**by:**

The *version* of a compilation unit changes whenever the compilation unit changes in a semantically significant way. This International Standard does not define the exact meaning of "semantically significant". It is also unspecified whether there are other events (such as recompilation) that result in the version of a compilation unit changing.

If P is not a library unit, and P has no completion, then P'Body_Version returns the Body_Version of the innermost program unit enclosing the declaration of P. If P is a library unit, and P has no completion, then P'Body_Version returns a value that is different from Body_Version of any version of P that has a completion.

## E.4 Remote Subprogram Calls

**Replace paragraph 18: [8652/0085]**

In a remote subprogram call with a formal parameter of a class-wide type, a check is made that the tag of the actual parameter identifies a tagged type declared in a declared-pure or shared passive library unit, or in the visible part of a remote types or remote call interface library unit. Program_Error is raised if this check fails.

**by:**

In a remote subprogram call with a formal parameter of a class-wide type, a check is made that the tag of the actual parameter identifies a tagged type declared in a declared-pure or shared passive library unit, or in the visible part of a remote types or remote call interface library unit. Program_Error is raised if this check fails. In a remote function call which returns a class-wide type, the same check is made on the function result.

**Insert after paragraph 20: [8652/0086]**

The implementation of remote subprogram calls shall conform to the PCS interface as defined by the specification of the language-defined package System.RPC (see E.5). The calling stub shall use the Do_RPC procedure unless the remote procedure call is asynchronous in which case Do_APC shall be used. On the receiving side, the corresponding receiving stub shall be invoked by the RPC-receiver.

**the new paragraph:**

With respect to shared variables in shared passive library units, the execution of the corresponding subprogram body of a synchronous remote procedure call is considered to be part of the execution of the calling task. The execution of the corresponding subprogram body of an asynchronous remote procedure call proceeds in parallel with the calling task and does not signal the next action of the calling task (see 9.10).

## E.5 Partition Communication Subsystem

**Insert after paragraph 24: [8652/0087]**

The implementation of the RPC-receiver shall be reentrant, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition.

**the new paragraphs:**

An implementation shall not restrict the replacement of the body of System.RPC. An implementation shall not restrict children of System.RPC. The related implementation permissions in the introduction to Annex A do not apply.

If the implementation of System.RPC is provided by the user, an implementation shall support remote subprogram calls as specified.

# Annex F: Information Systems

## F.3.1 Picture String Formation

**Replace paragraph 43: [8652/0088]**

- If a picture String has '+' or '-' as fixed_LHS_sign, in a floating_LHS_sign, or in an all_sign_number, then it has no RHS_sign.

**by:**

- If a picture String has '+' or '-' as fixed_LHS_sign, in a floating_LHS_sign, or in an all_sign_number, then it has no RHS_sign or '>' character.

## F.3.2 Edited Output Generation

**Replace paragraph 74: [8652/0089]**

```
123456.78     Picture:  "-$$$**_***_**9.99"
              Result:   "bbb$***123,456.78"
                       "bbbFF***123.456,78" (currency = "FF",
                                             separator = '.',
                                             radix mark = ',')
```

**by:**

```
123456.78     Picture:    "-$**_***_**9.99"
              Result:     "b$***123,456.78"
                         "bFF***123.456,78" (currency = "FF",
                                             separator = '.',
                                             radix mark = ',')
```

# Annex G: Numerics

## G.1.1 Complex Types

**Replace paragraph 2: [8652/0090]**

```
generic
   type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types is
   pragma Pure(Generic_Complex_Types);
```

**by:**

```
generic
   type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types is
   pragma Pure(Generic_Complex_Types);
```

**Replace paragraph 25: [8652/0020]**

The library package Numerics.Complex_Types defines the same types, constants, and subprograms as Numerics.Generic_Complex_Types, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents of Numerics.Generic_Complex_Types for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Types, Numerics.Long_Complex_Types, etc.

**by:**

The library package Numerics.Complex_Types is declared pure and defines the same types, constants, and subprograms as Numerics.Generic_Complex_Types, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents of Numerics.Generic_Complex_Types for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Types, Numerics.Long_Complex_Types, etc.

**Replace paragraph 55: [8652/0091]**

Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent, when the exponent is positive, or dividing the argument by the absolute value of the given exponent, when the exponent is negative; and reconverting to a cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

**by:**

Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation, exponentiating the modulus by the given exponent, multiplying the argument by the given exponent, and reconverting to a cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

## G.1.2 Complex Elementary Functions

**Replace paragraph 9: [8652/0020]**

The library package Numerics.Complex_Elementary_Functions defines the same subprograms as Numerics.Generic_Complex_Elementary_Functions, except that the predefined type Float is systematically substituted for Real'Base, and the Complex and Imaginary types exported by Numerics.Complex_Types are systematically substituted for Complex and Imaginary, throughout.

Nongeneric equivalents of Numerics.Generic_Complex_Elementary_Functions corresponding to each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Elementary_Functions, Numerics.Long_Complex_Elementary_Functions, etc.

**by:**

The library package Numerics.Complex_Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Complex_Elementary_Functions, except that the predefined type Float is systematically substituted for Real'Base, and the Complex and Imaginary types exported by Numerics.Complex_Types are systematically substituted for Complex and Imaginary, throughout. Nongeneric equivalents of Numerics.Generic_Complex_Elementary_Functions corresponding to each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Elementary_Functions, Numerics.Long_Complex_Elementary_Functions, etc.

## G.1.3 Complex Input-Output

**Replace paragraph 12: [8652/0092]**

The input sequence is a pair of optionally signed real literals representing the real and imaginary components of a complex value; optionally, the pair of components may be separated by a comma and/or surrounded by a pair of parentheses. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter Width is zero, then

**by:**

The input sequence is a pair of optionally signed real values representing the real and imaginary components of a complex value. These components have the format defined for the corresponding Get procedure of an instance of Text_IO.Float_IO (see A.10.9) for the base subtype of Complex_Types.Real. The pair of components may be separated by a comma or surrounded by a pair of parentheses or both. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter Width is zero, then

# Annex H: Safety and Security

## H.3.2 Pragma Inspection_Point

**Replace paragraph 5: [8652/0093]**

An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments.

**by:**

An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments and the object is visible at the inspection point.

## H.4 Safety and Security Restrictions

**Replace paragraph 8: [8652/0042]**

No_Local_Allocators
Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies; instantiations of generic packages are also prohibited in these contexts.

**by:**

No_Local_Allocators
Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies.

# Annex J: Obsolescent Features

## J.7.1 Interrupt Entries

**Replace paragraph 16: [8652/0077]**

Interrupt entry calls may be implemented by having the hardware execute directly the appropriate accept body. Alternatively, the implementation is allowed to provide an internal interrupt handler to simulate the effect of a normal task calling the entry.

**by:**

Interrupt entry calls may be implemented by having the hardware directly execute the appropriate accept_statement. Alternatively, the implementation is allowed to provide an internal interrupt handler to simulate the effect of a normal task calling the entry.

**Replace paragraph 20: [8652/0077]**

NOTES

1 Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an accept body executed in response to an interrupt can be executed with the active priority at which the hardware generates the interrupt, taking precedence over lower priority tasks, without a scheduling action.

**by:**

NOTES

1 Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an accept_statement executed in response to an interrupt can be executed with the active priority at which the hardware generates the interrupt, taking precedence over lower priority tasks, without a scheduling action.