

Information technology — Programming languages — Ada

AMENDMENT 1 (Draft 4)

Technologies de l'information — Langages de programmation — Ada

AMENDEMENT 1

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

© 2003, AXE Consultants. All Rights Reserved.

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

Introduction

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

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

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

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

- The Ravenscar profile to provide a simplified tasking system for high-integrity systems (see clause D.13);
- A non-preemptive task dispatching policy (see clause D.2.4);
- Aggregates and constants for limited types (see clause 4.3.1 and 7.5);
- A mechanism for writing C unions to make interfaces with C systems easier (see clause B.3.3);
- A pragma to revoke the suppression of checks (see clause 11.5);
- A method of making compilation units visible only in the private part of a package (see clause 10.1.2); and
- File directory and name management functions (see clause A.16).

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

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

Typographical conventions:

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

Disclaimer:

This document is a draft of a possible amendment to Ada 95 (International Standard ISO/IEC 8652:1995). This draft contains only proposals substantially approved by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group (ARG). Many other important proposals are under consideration by the ARG. Neither the ARG nor any other group has determined which, if any, of these proposals will be included in the amendment. Any proposal may be substantially changed or withdrawn before this document begins standardization, and other proposals may be added. This document is not an official publication or work product of the ARG.

Section 1: General

No changes in this section.

Section 2: Lexical Elements

No changes in this section.

Section 3: Declarations and Types

3.3.1 Object Declarations

Replace paragraph 5: [AI95-00287-01]

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

by:

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

3.6.2 Operations of Array Types

Replace paragraph 16: [AI95-00287-01]

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

by:

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

3.8 Record Types

Delete paragraph 8: [AI95-00287-01]

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

Replace paragraph 25: [AI95-00287-01]

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

by:

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

3.9.2 Dispatching Operations of Tagged Types

Replace paragraph 17: [AI95-00196-01]

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

by:

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

Insert after paragraph 18: [AI95-00196-01]

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

the new paragraph:

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

3.10 Access Types

Replace paragraph 9: [AI95-00225-01]

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

by:

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

3.10.2 Operations of Access Types

Replace paragraph 2: [AI95-00235-01]

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

by:

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

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

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

Replace paragraph 32: [AI95-00229-01]

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

by:

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

Section 4: Names and Expressions

4.3 Aggregates

Replace paragraph 3: [AI95-00287-01]

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

by:

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

4.3.1 Record Aggregates

Replace paragraph 4: [AI95-00287-01]

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

by:

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

Replace paragraph 8: [AI95-00287-01]

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

by:

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

Replace paragraph 16: [AI95-00287-01]

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

by:

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

Insert after paragraph 17: [AI95-00287-01]

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

the new paragraph:

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

Insert before paragraph 20: [AI95-00287-01]

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

the new paragraph:

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

4.3.2 Extension Aggregates

Replace paragraph 4: [AI95-00287-01]

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

by:

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

Replace paragraph 5: [AI95-00306-01]

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

by:

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

4.3.3 Array Aggregates

Replace paragraph 3: [AI95-00287-01]

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

by:

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

Replace paragraph 5: [AI95-00287-01]

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

by:

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

Replace paragraph 7: [AI95-00287-01]

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

by:

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

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

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

the new paragraph:

Each array component expression defines the value for the associated component(s). For an component given by $\langle \rangle$, the associated component(s) are initialized by default (see 3.3.1).

4.6 Type Conversions

Replace paragraph 9: [AI95-00246-01]

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

by:

If the target type is an array type, then the operand type shall be an array type. The target type and operation type shall have a common ancestor, or:

Replace paragraph 12: [AI95-00246-01]

- The component subtypes shall statically match; and

by:

- The component subtypes shall statically match;

Replace paragraph 12.1: [AI95-00246-01]

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

by:

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

4.8 Allocators

Replace paragraph 5: [AI95-00287-01]

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

by:

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

4.9 Static Expressions and Static Subtypes

Replace paragraph 26: [AI95-00263-01]

A *static subtype* is either a *static scalar subtype* or a *static string subtype*. A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal scalar type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. A static string subtype

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

by:

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

Replace paragraph 38: [AI95-00268-01]

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

by:

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

Implementation Advice

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

Section 5: Statements

No changes in this section.

Section 6: Subprograms

6.4 Subprogram Calls

Replace paragraph 8: [AI95-00310-01]

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

by:

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

6.5 Return Statements

Replace paragraph 18: [AI95-00316-01]

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

by:

- a name that denotes an object view (or a value with an associated object, see 6.2) whose accessibility level is not deeper than that of the master that elaborated the function body; or

Section 7: Packages

7.4 Deferred Constants

Replace paragraph 9: [AI95-00256-01]

The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4).

by:

The completion of a deferred constant declaration shall occur before the constant is frozen (see 13.14).

7.5 Limited Types

Replace paragraph 1: [AI95-00287-01]

A limited type is (a view of) a type for which the assignment operation is not allowed. A nonlimited type is a (view of a) type for which the assignment operation is allowed.

by:

A limited type is (a view of) a type for which copying (such as for an assignment_statement) is not allowed. A nonlimited type is a (view of a) type for which copying is allowed.

Insert before paragraph 2: [AI95-00287-01]

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

the new paragraph:

For an assignment operation that initializes a limited object with the value of an expression, the expression shall be a (possibly parenthesized or qualified) aggregate.

Insert after paragraph 8: [AI95-00287-01]

There are no predefined equality operators for a limited type.

the new paragraph:

Implementation Requirements

For an aggregate of a limited type used to initialize an object as allowed above, the implementation shall not create a separate anonymous object for the aggregate. The aggregate shall be constructed directly in the new object.

Replace paragraph 9: [AI95-00287-01]

13 The following are consequences of the rules for limited types:

by:

13 While limited types have an assignment operation, other rules of the language insure that it is never actually invoked. The source of such an assignment operation must be an aggregate, and such aggregates must be built directly in the target object.

Delete paragraph 10: [AI95-00287-01]

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

Delete paragraph 11: [AI95-00287-01]

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

Delete paragraph 12: [AI95-00287-01]

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

Delete paragraph 13: [AI95-00287-01]

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

Delete paragraph 14: [AI95-00287-01]

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

Delete paragraph 15: [AI95-00287-01]

15 The rules do not exclude a `default_expression` for a formal parameter of a limited type; they do not exclude a deferred constant of a limited type if the full declaration of the constant is of a nonlimited type.

7.6 User-Defined Assignment and Finalization**Replace paragraph 5: [AI95-00161-01]**

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

by:

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

Replace paragraph 7: [AI95-00161-01]

```
type Limited_Controlled is abstract tagged limited private;
```

by:

```
type Limited_Controlled is abstract tagged limited private;
pragma Preelaborable_Initialization(Limited_Controlled);
```

Replace paragraph 21: [AI95-00147-01]

- For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an `assignment_statement`, the implementation need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object). Even if an anonymous object is created, the implementation may move its value to the target object as part of the assignment without re-adjusting so long as the anonymous object has no aliased subcomponents.

by:

- For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an `assignment_statement`, the implementation

need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

Furthermore, an implementation is permitted to omit implicit Initialize, Adjust, and Finalize calls and associated assignment operations on an object of nonlimited controlled type provided that:

- any omitted Initialize call is not a call on a user-defined Initialize procedure, and
- any usage of the value of the object after the implicit Initialize or Adjust call and before any subsequent Finalize call on the object does not change the external effect of the program, and
- after the omission of such calls and operations, any execution of the program that executes an Initialize or Adjust call on an object or initializes an object by an **aggregate** will also later execute a Finalize call on the object and will always do so prior to assigning a new value to the object, and
- the assignment operations associated with omitted Adjust calls are also omitted.

This permission applies to Adjust and Finalize calls even if the implicit calls have additional external effects.

7.6.1 Completion and Finalization

Replace paragraph 16: [AI95-00256-01]

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

by:

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

Section 8: Visibility Rules

8.3 Visibility

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

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

the new paragraph:

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

8.5.4 Subprogram Renaming Declarations

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

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

the new paragraph:

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

Section 9: Tasks and Synchronization

9.1 Task Units and Task Objects

Replace paragraph 21: [AI95-00287-01]

4 A task type is a limited type (see 7.5), and hence has neither an assignment operation nor predefined equality operators. If an application needs to store and exchange task identities, it can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7).

by:

4 A task type is a limited type (see 7.5), and hence has neither assignment nor predefined equality operators. If an application needs to store and exchange task identities, it can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7).

9.4 Protected Units and Protected Objects

Replace paragraph 23: [AI95-00287-01]

15 A protected type is a limited type (see 7.5), and hence has neither an assignment operation nor predefined equality operators.

by:

15 A protected type is a limited type (see 7.5), and hence has neither assignment nor predefined equality operators.

9.6 Delay Statements, Duration, and Time

Replace paragraph 10: [AI95-00161-01]

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

by:

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

Section 10: Program Structure and Compilation Issues

10.1.2 Context Clauses - With Clauses

Replace paragraph 4: [AI95-00262-01]

`with_clause ::= with library_unit_name {, library_unit_name}`

by:

`with_clause ::= [private] with library_unit_name {, library_unit_name}`

Replace paragraph 8: [AI95-00220-01; AI95-00262-01]

If a `with_clause` of a given `compilation_unit` mentions a private child of some library unit, then the given `compilation_unit` shall be either the declaration of a private descendant of that library unit or the body or a subunit of a (public or private) descendant of that library unit.

by:

If a `with_clause` of a given `compilation_unit` mentions a private child of some library unit, then the given `compilation_unit` shall be one of:

- the declaration, body, or subunit of a private descendant of that library unit;
- the body or subunit of a public descendant of that library unit, but not a subprogram body acting as a subprogram declaration (see 10.1.4); or
- the declaration of a public descendant of that library unit, and the `with_clause` shall include the keyword **private**.

A name denoting a library item that is visible only due to being mentioned in `with_clauses` that include the keyword **private** shall appear only within

- a private part,
- a body, but not within the `subprogram_specification` of a library subprogram body,
- a private descendant of the unit on which one of these `with_clauses` appear, or
- a pragma within a context clause.

10.1.3 Subunits of Compilation Units

Replace paragraph 8: [AI95-00243-01]

The *parent body* of a subunit is the body of the program unit denoted by its `parent_unit_name`. The term *subunit* is used to refer to a subunit and also to the `proper_body` of a subunit.

by:

The *parent body* of a subunit is the body of the program unit denoted by its `parent_unit_name`. The term *subunit* is used to refer to a subunit and also to the `proper_body` of a subunit. A *subunit of a program unit* includes subunits declared directly in the program unit as well as any subunits declared in those subunits (recursively).

10.1.5 Pragmas and Program Units

Replace paragraph 9: [AI95-00212-01]

An implementation may place restrictions on configuration pragmas, so long as it allows them when the environment contains no `library_items` other than those of the predefined environment.

by:

An implementation may require that configuration pragmas that select partition-wide or system-wide options be compiled when the environment contains no `library_item`s other than those of the predefined environment. In this case, the implementation must still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options.

10.2 Program Execution

Replace paragraph 9: [AI95-00256-01]

The order of elaboration of library units is determined primarily by the *elaboration dependences*. There is an elaboration dependence of a given `library_item` upon another if the given `library_item` or any of its subunits depends semantically on the other `library_item`. In addition, if a given `library_item` or any of its subunits has a pragma `Elaborate` or `Elaborate_All` that mentions another library unit, then there is an elaboration dependence of the given `library_item` upon the body of the other library unit, and, for `Elaborate_All` only, upon each `library_item` needed by the declaration of the other library unit.

by:

The order of elaboration of library units is determined primarily by the *elaboration dependences*. There is an elaboration dependence of a given `library_item` upon another if the given `library_item` or any of its subunits depends semantically on the other `library_item`. In addition, if a given `library_item` or any of its subunits has a pragma `Elaborate` or `Elaborate_All` that names another library unit, then there is an elaboration dependence of the given `library_item` upon the body of the other library unit, and, for `Elaborate_All` only, upon each `library_item` needed by the declaration of the other library unit.

10.2.1 Elaboration Control

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

A pragma `Preelaborate` is a library unit pragma.

the new paragraphs:

The form of pragma `Preelaborable_Initialization` is as follows:

```
pragma Preelaborable_Initialization (direct_name);
```

Replace paragraph 9: [AI95-00161-01]

- The creation of a default-initialized object (including a component) of a descendant of a private type, private extension, controlled type, task type, or protected type with `entry_declarations`; similarly the evaluation of an `extension_aggregate` with an ancestor `subtype_mark` denoting a subtype of such a type.

by:

- The creation of an object (including a component) of a type which does not have preelaborable initialization. Similarly the evaluation of an `extension_aggregate` with an ancestor `subtype_mark` denoting a subtype of such a type.

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

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

the new paragraphs:

The following rules specify which entities have preelaborable initialization:

- The partial view of a private type or private extension, a protected type without `entry_declarations`, a generic formal private type, or a generic formal derived type, have preelaborable initialization if and only if the `pragma Preelaborable_Initialization` has been applied to them.
- A component (including a discriminant) of a record or protected type has preelaborable initialization if its declaration includes a `default_expression` whose execution does not perform any actions prohibited in preelaborable constructs as described above, or if its declaration does not include a default expression and its type has preelaborable initialization.
- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a derived record or protected type) if the non-inherited components all have preelaborable initialization. Moreover, a user-defined controlled type with an overriding `Initialize` procedure does not have preelaborable initialization.
- A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, or a record type whose components all have preelaborable initialization.

A `pragma Preelaborable_Initialization` specifies that a type has preelaborable initialization. This pragma shall appear in the visible part of a package or generic package.

If the pragma appears in the first list of `declarative_items` of a `package_specification`, then the `direct_name` shall denote the first subtype of a private type, private extension, or protected type without `entry_declarations`, and the type shall be declared within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, each component of the protected type shall have preelaborable initialization. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit.

If the pragma appears in a `generic_formal_part`, then the `direct_name` shall denote a generic formal private type or a generic formal derived type declared in the same `generic_formal_part` as the pragma. In a `generic_instantiation` the corresponding actual type shall have preelaborable initialization.

Section 11: Exceptions

11.4.1 The Package Exceptions

Replace paragraph 14: [AI95-00241-01]

Raise_Exception and Reraise_Occurrence have no effect in the case of Null_Id or Null_Occurrence. Exception_Message, Exception_Identity, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Id or Null_Occurrence.

by:

Raise_Exception and Reraise_Occurrence have no effect in the case of Null_Id or Null_Occurrence. Exception_Name raises Constraint_Error for a Null_Id. Exception_Message, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence. Exception_Identity applied to Null_Occurrence returns Null_Id.

11.5 Suppressing Checks

Replace paragraph 1: [AI95-00224-01]

A pragma Suppress gives permission to an implementation to omit certain language-defined checks.

by:

Checking pragmas give an implementation instructions on handling language-defined checks. A pragma Suppress gives permission to an implementation to omit certain language-defined checks, while a pragma Unsuppress revokes the permission to omit checks.

Replace paragraph 3: [AI95-00224-01]

The form of a pragma Suppress is as follows:

by:

The forms of checking pragmas are as follows:

Replace paragraph 4: [AI95-00224-01]

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

by:

pragma Suppress(identifier);

pragma Unsuppress(identifier);

Replace paragraph 5: [AI95-00224-01]

A pragma Suppress is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.

by:

A checking pragma is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.

Replace paragraph 6: [AI95-00224-01]

The identifier shall be the name of a check. The name (if present) shall statically denote some entity.

by:

The identifier shall be the name of a check.

Delete paragraph 7: [AI95-00224-01]

For a pragma Suppress that is immediately within a `package_specification` and includes a `name`, the `name` shall denote an entity (or several overloaded subprograms) declared immediately within the `package_specification`.

Replace paragraph 8: [AI95-00224-01]

A pragma Suppress gives permission to an implementation to omit the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a `package_specification` and includes a `name`, to the end of the scope of the named entity. If the pragma includes a `name`, the permission applies only to checks performed on the named entity, or, for a subtype, on objects and values of its type. Otherwise, the permission applies to all entities. If permission has been given to suppress a given check, the check is said to be *suppressed*.

by:

A checking pragma applies to the named check in a specific region (see below), and applies to all entities in that region. A checking pragma given in a `declarative_part` or immediately within a `package_specification` applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a checking pragma given as a configuration pragma is the declarative region for the entire compilation unit to which it applies.

If a checking pragma applies to a generic instantiation, then the checking pragma also applies to the instance. If a checking pragma applies to a call to a subprogram that has a pragma Inline applied to it, then the checking pragma also applies to the inlined subprogram body.

A pragma Suppress gives permission to an implementation to omit the named check (or every check in the case of `All_Checks`) for any entities to which it applies. If permission has been given to suppress a given check, the check is said to be *suppressed*.

A pragma Unsuppress revokes the permission to omit the named check (or every check in the case of `All_Checks`) given by any pragma Suppress that applies at the point of the pragma Unsuppress. The permission is revoked for the region to which the pragma Unsuppress applies. If there is no such permission at the point of a pragma Unsuppress, then the pragma has no effect. A later pragma Suppress can renew the permission.

Replace paragraph 27: [AI95-00224-01]

An implementation is allowed to place restrictions on Suppress pragmas. An implementation is allowed to add additional check names, with implementation-defined semantics. When `Overflow_Check` has been suppressed, an implementation may also suppress an unspecified subset of the `Range_Checks`.

by:

An implementation is allowed to place restrictions on checking pragmas, subject only to the requirement that pragma Unsuppress shall allow any check names supported by pragma Suppress. An implementation is allowed to add additional check names, with implementation-defined semantics. When `Overflow_Check` has been suppressed, an implementation may also suppress an unspecified subset of the `Range_Checks`.

An implementation may support an additional parameter on pragma Unsuppress similar to the one allowed for pragma Suppress (see J.10). The meaning of such a parameter is implementation-defined.

Insert after paragraph 29: [AI95-00224-01]

2 There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons.

the new paragraph:

3 It is possible to give both a pragma Suppress and Unsuppress for the same check immediately within the same `declarative_part`. In that case, the last pragma given determines whether or not the check is suppressed.

Similarly, it is possible to resuppress a check which has been unsuppressed by giving a pragma Suppress in an inner declarative region.

Section 12: Generic Units

12.4 Formal Objects

Delete paragraph 8: [AI95-00287-01]

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

Replace paragraph 9: [AI95-00255-01]

A `formal_object_declaration` declares a generic formal object. The default mode is **in**. For a formal object of mode **in**, the nominal subtype is the one denoted by the `subtype_mark` in the declaration of the formal. For a formal object of mode **in out**, its type is determined by the `subtype_mark` in the declaration; its nominal subtype is nonstatic, even if the `subtype_mark` denotes a static subtype.

by:

A `formal_object_declaration` declares a generic formal object. The default mode is **in**. For a formal object of mode **in**, the nominal subtype is the one denoted by the `subtype_mark` in the declaration of the formal. For a formal object of mode **in out**, its type is determined by the `subtype_mark` in the declaration; its nominal subtype is nonstatic, even if the `subtype_mark` denotes a static subtype; for a composite type, its nominal subtype is unconstrained if the first subtype of the type is unconstrained, even if the `subtype_mark` denotes a constrained subtype.

12.5 Formal Types

Replace paragraph 8: [AI95-00233-01]

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

by:

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

12.5.1 Formal Private and Derived Types

Replace paragraph 20: [AI95-00233-01]

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new `discriminant_part` is not specified), as for a derived type defined by a `derived_type_definition` (see 3.4).

by:

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new `discriminant_part` is not specified), as for a derived type defined by a `derived_type_definition` (see 3.4 and 7.3.1).

Replace paragraph 21: [AI95-00233-01]

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

by:

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

Section 13: Representation Issues

13.3 Representation Attributes

Delete paragraph 26: [AI95-00247-01]

If an Alignment is specified for a composite subtype or object, this Alignment shall be equal to the least common multiple of any specified Alignments of the subcomponent subtypes, or an integer multiple thereof.

13.7 The Package System

Replace paragraph 12: [AI95-00161-01]

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

by:

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

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

```
Default_Bit_Order : constant Bit_Order;
```

by:

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

Replace paragraph 35: [AI95-00221-01]

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order.

by:

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

13.9.1 Data Validity

Replace paragraph 12: [AI95-00167-01]

A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, and the result object has an invalid representation.

by:

A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, the result object has an invalid representation, and the result is used other than as the **expression** of an **assignment_statement** or an object declaration, or as the prefix of a Valid attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a "Valid" attribute reference, is erroneous.

13.11 Storage Management

Replace paragraph 6: [AI95-00161-01]

```
type Root_Storage_Pool is
```

```
abstract new Ada.Controlled.Limited_Controlled with private;
```

by:

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

13.11.1 The Max_Size_In_Storage_Elements Attribute

Replace paragraph 3: [AI95-00256-01]

Denotes the maximum value for Size_In_Storage_Elements that will be requested via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

by:

Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

13.12 Pragma Restrictions

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

The set of restrictions is implementation defined.

the new paragraphs:

The following *restriction_identifiers* are language-defined (additional restrictions are defined in the Specialized Needs Annexes):

No_Implementation_Attributes

There are no implementation-defined attributes. This restriction applies only to the current compilation or environment, not the entire partition.

No_Implementation_Pragmas

There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

13.13.1 The Package Streams

Replace paragraph 3: [AI95-00161-01]

```
type Root_Stream_Type is abstract tagged limited private;
```

by:

```
type Root_Stream_Type is abstract tagged limited private;
pragma Preelaborable_Initialization(Root_Stream_Type);
```

Replace paragraph 8: [AI95-00227-01]

The Read operation transfers Item'Length stream elements from the specified stream to fill the array Item. The index of the last stream element transferred is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

by:

The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any

elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

Insert after paragraph 10: [AI95-00227-01]

See A.12.1, "The Package Streams.Stream_IO" for an example of extending type Root_Stream_Type.

the new paragraph:

If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error.

13.13.2 Stream-Oriented Attributes

Insert before paragraph 2: [AI95-00270-01]

For every subtype *S* of a specific type *T*, the following attributes are defined.

the new paragraphs:

For every subtype *S* of an elementary type *T*, the following operational attribute is defined:

S'Stream_Size

Denotes the number of bits occupied in a stream by items of subtype *S*. Hence, the number of stream elements required per item of elementary type *T* is:

$$T'Stream_Size / Ada.Streams.Stream_Element'Size$$

The value of this attribute is type universal_integer and a multiple of Stream_Element'Size.

Stream_Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, non-negative, and a multiple of Stream_Element'Size.

Implementation Advice

The recommended level of support for the Stream_Size attribute is: A Stream_Size clause should be supported for an elementary type *T* if the specified Stream_Size is a multiple of Stream_Element'Size and is no less than the size of the first subtype of *T*, and no greater than the size of the largest type of the same elementary class (signed integer, modular integer, float, ordinary fixed, decimal fixed, or access).

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

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

by:

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

Constraint_Error is raised by the predefined Write attribute if the value of the elementary item is outside the range of values representable using Stream_Size bits. For a signed integer type, an enumeration type, or a fixed-point type, the range is unsigned only if the integer code for the first subtype low bound is non-negative, and a (symmetric) signed range that covers all values of the first subtype would require more than Stream_Size bits; otherwise the range is signed.

Replace paragraph 17: [AI95-00270-01]

If a stream element is the same size as a storage element, then the normal in-memory representation should be used by Read and Write for scalar objects. Otherwise, Read and Write should use the smallest number of stream elements needed to represent all values in the base range of the scalar type.

by:

By default, the predefined stream-oriented attributes for an elementary type should only read or write the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

Replace paragraph 27: [AI95-00195-01]

S'Output then calls S'Write to write the value of Item to the stream. S'Input then creates an object (with the bounds or discriminants, if any, taken from the stream), initializes it with S'Read, and returns the value of the object.

by:

S'Output then calls S'Write to write the value of Item to the stream. S'Input then creates an object (with the bounds or discriminants, if any, taken from the stream), passes it to S'Read, and returns the value of the object. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, 7.6.1).

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

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

the new paragraphs:

S'Class"Tag_Write

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

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

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

S'Class"Tag_Read

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

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

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

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

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

Replace paragraph 31: [AI95-00260-01]

First writes the external tag of *Item* to *Stream* (by calling `String'Output(Tags.External_Tag(Item'Tag))` -- see 3.9) and then dispatches to the subprogram denoted by the `Output` attribute of the specific type identified by the tag.

by:

First writes the external tag of *Item* to *Stream* (by calling `S'Tag_Write(Stream, Item'Tag)`) and then dispatches to the subprogram denoted by the `Output` attribute of the specific type identified by the tag.

Replace paragraph 34: [AI95-00260-01]

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling `Tags.Internal_Tag(String'Input(Stream))` -- see 3.9) and then dispatches to the subprogram denoted by the `Input` attribute of the specific type identified by the internal tag; returns that result.

by:

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling `S'Tag_Read(Stream)`) and then dispatches to the subprogram denoted by the `Input` attribute of the specific type identified by the internal tag; converts that result to `S'Class` and returns it.

Replace paragraph 35: [AI95-00195-01]

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

by:

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

It is unspecified at which point and in which order these checks are performed. In particular, if `Constraint_Error` is raised due to the failure of one of these checks, it is unspecified how many stream elements have been read from the stream.

Replace paragraph 36: [AI95-00195-01]

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

by:

The stream-oriented attributes may be specified for any type via an `attribute_definition_clause`. The subprogram name given in such a clause shall not denote an abstract subprogram.

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

- The attribute_designator is Read, Write or Output, and *T* is nonlimited.
- The attribute_designator is Input, and *T* is nonlimited and not abstract.
- The attribute_designator is Read (resp. Write) and *T* is a limited record extension, and the attribute Read (resp. Write) is available for the parent type of *T* and for the types of all of the extension components.
- The attribute_designator is Input (resp. Output), and *T* is a limited type, and the attribute Read (resp. Write) is available for *T*.
- The attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible.

A stream-oriented attribute for a subtype of a class-wide type T'Class is available at places where one of the following conditions is true:

- *T* is nonlimited; or
- The attribute has been specified via an attribute_definition_clause, and the attribute_definition_clause is visible; or
- where the corresponding attribute of *T* is available, provided that if *T* has a partial view, the corresponding attribute is available at the end of the visible part where *T* is declared.

An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference.

In the parameter_and_result_profiles for the stream-oriented attributes, the subtype of the Item parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. The same rule applies to the result of the Input attribute.

For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

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

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

the new paragraphs:

If Constraint_Error is raised during a call to Read because of failure of one of the above checks, the implementation must ensure that the discriminants of the actual parameter of Read are not modified.

Implementation Permissions

The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. However, all the calls on the Read and Write operations of the stream type needed to implement an explicit invocation of a stream-oriented attribute must take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see 12.3).

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

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

the new paragraph:

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

Annex A: Predefined Language Environment

A.4.2 The Package Strings.Maps

Replace paragraph 4: [AI95-00161-01]

```
-- Representation for a set of Wide_Character values:  
type Wide_Character_Set is private;
```

by:

```
-- Representation for a set of Wide_Character values:  
type Wide_Character_Set is private;  
pragma Preelaborable_Initialization(Wide_Character_Set);
```

Replace paragraph 4: [AI95-00161-01]

```
-- Representation for a set of character values:  
type Character_Set is private;
```

by:

```
-- Representation for a set of character values:  
type Character_Set is private;  
pragma Preelaborable_Initialization(Character_Set);
```

Replace paragraph 20: [AI95-00161-01]

```
-- Representation for a Wide_Character to Wide_Character mapping:  
type Wide_Character_Mapping is private;
```

by:

```
-- Representation for a Wide_Character to Wide_Character mapping:  
type Wide_Character_Mapping is private;  
pragma Preelaborable_Initialization(Wide_Character_Mapping);
```

Replace paragraph 20: [AI95-00161-01]

```
-- Representation for a character to character mapping:  
type Character_Mapping is private;
```

by:

```
-- Representation for a character to character mapping:  
type Character_Mapping is private;  
pragma Preelaborable_Initialization(Character_Mapping);
```

A.4.4 Bounded-Length String Handling

Replace paragraph 101: [AI95-00238-01]

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

by:

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

A.4.5 Unbounded-Length String Handling

Replace paragraph 4: [AI95-00161-01]

```
type Unbounded_String is private;
```

by:

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

A.5.3 Attributes of Floating Point Types

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

The function yields the integral value nearest to X , rounding toward the even integer if X lies exactly halfway between two integers. A zero result has the sign of X when $S'Signed_Zeros$ is True.

the new paragraphs:

$S'Machine_Rounding$

$S'Machine_Rounding$ denotes a function with the following specification:

```
function S'Machine_Rounding (X : T)
return T
```

The function yields the integral value nearest to X . If X lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of X when $S'Signed_Zeros$ is True. This function provides access to the rounding behavior which is most efficient on the target processor.

A.8 Sequential and Direct Files

Replace paragraph 1: [AI95-00283-01]

Two kinds of access to external files are defined in this subclause: *sequential access* and *direct access*. The corresponding file types and the associated operations are provided by the generic packages `Sequential_IO` and `Direct_IO`. A file object to be used for sequential access is called a *sequential file*, and one to be used for direct access is called a *direct file*. Access to stream files is described in A.12.1.

by:

Two kinds of access to external files are defined in this subclause: *sequential access* and *direct access*. The corresponding file types and the associated operations are provided by the generic packages `Sequential_IO` and `Direct_IO`. A file object to be used for sequential access is called a *sequential file*, and one to be used for direct access is called a *direct file*. Access to *stream files* is described in A.12.1.

A.8.2 File Management

Replace paragraph 3: [AI95-00283-01]

Establishes a new external file, with the given name and form, and associates this external file with the given file. The given file is left open. The current mode of the given file is set to the given access mode. The default access mode is the mode `Out_File` for sequential and text input-output; it is the mode `Inout_File` for direct input-output. For direct access, the size of the created file is implementation defined.

by:

Establishes a new external file, with the given name and form, and associates this external file with the given file. The given file is left open. The current mode of the given file is set to the given access mode. The default

access mode is the mode Out_File for sequential, stream, and text input-output; it is the mode Inout_File for direct input-output. For direct access, the size of the created file is implementation defined.

Replace paragraph 22: [AI95-00248-01]

Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an Open operation). If an external environment allows alternative specifications of the name (for example, abbreviations), the string returned by the function should correspond to a full specification of the name.

by:

Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an Open operation).

A.10.6 Get and Put Procedures

In paragraph 5 replace: [AI95-00223-01]

Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. Get procedures for numeric or enumeration types start by skipping leading blanks, where a *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

by:

Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. A *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

A.12.1 The Package Streams.Stream_IO

Replace paragraph 28: [AI95-00283-01]

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

by:

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

The End_of_File function:

- Propagates Mode_Error if the mode of the file is not In_File;
- If positioning is supported for the given external file, the function returns True if the current index exceeds the size of the external file; otherwise it returns False;
- Otherwise, the function returns True if no more elements can be read from the given file; otherwise it returns False.

Replace paragraph 28.1: [AI95-00085-01]

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

by:

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

Replace paragraph 30: [AI95-00256-01]

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

by:

The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index. For a file that supports positioning, Read without a Positive_Count parameter starts reading at the current index, and Write without a Positive_Count parameter starts writing at the current index.

A.16 The Package Directories

Insert new clause: [AI95-00248-01]

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

Static Semantics

The library package Ada.Directories has the following declaration:

```
with Ada.IO_Exceptions;
with Ada.Calendar;
package Ada.Directories is

    -- Directory and file operations:

    function Current_Directory return String;
    procedure Set_Directory (Directory : in String);

    procedure Create_Directory (New_Directory : in String;
                                Form : in String := "");

    procedure Delete_Directory (Directory : in String);

    procedure Create_Path (New_Directory : in String;
                           Form : in String := "");

    procedure Delete_Tree (Directory : in String);

    procedure Delete_File (Name : in String);

    procedure Rename (Old_Name, New_Name : in String);

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

    -- File and directory name operations:

    function Full_Name (Name : in String) return String;
```

```

function Simple_Name (Name : in String) return String;

function Containing_Directory (Directory : in String) return String;

function Extension (Name : in String) return String;

function Base_Name (Name : in String) return String;

function Compose (Containing_Directory : in String := "";
                  Name : in String;
                  Extension : in String := "") return String;

-- File and directory queries:

type File_Kind is (Directory, Ordinary_File, Special_File);

type File_Size is range 0 .. implementation-defined;

function Exists (Name : in String) return Boolean;

function Kind (Name : in String) return File_Kind;

function Size (Name : in String) return File_Size;

function Modification_Time (Name : in String) return Ada.Calendar.Time;

-- Directory searching:

type Directory_Entry_Type is limited private;

type Filter_Type is array (File_Kind) of Boolean;

type Search_Type is limited private;

procedure Start_Search (Search : in out Search_Type;
                       Directory : in String;
                       Pattern : in String;
                       Filter : in Filter_Type := (others => True));

procedure End_Search (Search : in out Search_Type);

function More_Entries (Search : in Search_Type) return Boolean;

procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);

-- Operations on Directory Entries:

function Simple_Name (Directory_Entry : in Directory_Entry_Type)
return String;

function Full_Name (Directory_Entry : in Directory_Entry_Type)
return String;

function Kind (Directory_Entry : in Directory_Entry_Type)
return File_Kind;

```

```

function Size (Directory_Entry : in Directory_Entry_Type)
    return File_Size;

function Modification_Time (Directory_Entry : in Directory_Entry_Type)
    return Ada.Calendar.Time;

Status_Error : exception renames Ada.IO_Exceptions.Status_Error;
Name_Error : exception renames Ada.IO_Exceptions.Name_Error;
Use_Error : exception renames Ada.IO_Exceptions.Use_Error;
Device_Error : exception renames Ada.IO_Exceptions.Device_Error;

private
    -- Not specified by the language.
end Ada.Directories;

```

External files may be classified as directories, special files, or ordinary files. A *directory* is an external file that is a container for files on the target system. A *special file* is an external file that cannot be created or read by a predefined Ada Input-Output package. External files that are not special files or directories are called *ordinary files*.

A *file name* is a string identifying an external file. Similarly, a *directory name* is a string identifying a directory. The interpretation of file names and directory names is implementation-defined.

The *full name* of an external file is a full specification of the name of the file. If the external environment allows alternative specifications of the name (for example, abbreviations), the full name should not use such alternatives. A full name typically will include the names of all of directories that contain the item. The *simple name* of an external file is the name of the item, not including any containing directory names. Unless otherwise specified, a file name or directory name parameter to a predefined Ada input-output subprogram can be a full name, a simple name, or any other form of name supported by the implementation.

The *default directory* is the directory that is used if a directory or file name is not a full name (that is, when the name does not fully identify all of the containing directories).

A *directory entry* is a single item in a directory, identifying a single external file (including directories and special files).

For each function that returns a string, the lower bound of the returned value is 1.

The following file and directory operations are provided:

```

function Current_Directory return String;

```

Returns the full directory name for the current default directory. The name returned shall be suitable for a future call to Set_Directory. The exception Use_Error is propagated if a default directory is not supported by the external environment.

```

procedure Set_Directory (Directory : in String);

```

Sets the current default directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support making Directory (in the absence of Name_Error) a default directory.

```

procedure Create_Directory (New_Directory : in String;
    Form : in String := "");

```

Creates a directory with name New_Directory. The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of a directory. The exception Use_Error is propagated if the external

environment does not support the creation of a directory with the given name (in the absence of Name_Error) and form.

procedure Delete_Directory (Directory : **in** String);

Deletes an existing empty directory with name Directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory (or some portion of its contents) with the given name (in the absence of Name_Error).

procedure Create_Path (New_Directory : **in** String;
Form : **in** String := "");

Creates zero or more directories with name New_Directory. Each non-existent directory named by New_Directory is created. For example, on a typical Unix system, Create_Tree ("/usr/me/my"); would create directory "me" in directory "usr", then create directory "my" in directory "me". The Form can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of any directory. The exception Use_Error is propagated if the external environment does not support the creation of any directories with the given name (in the absence of Name_Error) and form.

procedure Delete_Tree (Directory : **in** String);

Deletes an existing directory with name Directory. The directory and all of its contents (possibly including other directories) are deleted. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory or some portion of its contents with the given name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified if a portion of the contents of the directory are deleted.

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

Deletes an existing ordinary or special file with Name. The exception Name_Error is propagated if the string given as Name does not identify an existing ordinary or special external file. The exception Use_Error is propagated if the external environment does not support the deletion of the file with the given name (in the absence of Name_Error).

procedure Rename (Old_Name, New_Name : **in** String);

Renames an existing external file (including directories) with Old_Name to New_Name. The exception Name_Error is propagated if the string given as Old_Name does not identify an existing external file. The exception Use_Error is propagated if the external environment does not support the renaming of the file with the given name (in the absence of Name_Error). In particular, Use_Error is propagated if a file or directory already exists with New_Name.

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

Copies the contents of the existing external file with Source_Name to Target_Name. The resulting external file is a duplicate of the source external file. The Form can be used to give system-dependent characteristics of the resulting external file; the interpretation of the Form parameter is implementation-defined. Exception Name_Error is propagated if the string given as Source_Name does not identify an existing external ordinary or special file or if the string given as Target_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external environment does not support the creating of the file with the name given by Target_Name and form given by Form, or copying of the file with the name given by Source_Name (in the absence of Name_Error).

The following file and directory name operations are provided:

function Full_Name (Name : **in** String) **return** String;

Returns the full name corresponding to the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Simple_Name (Name : in String) return String;
```

Returns the simple name portion of the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Containing_Directory (Name : in String) return String;
```

Returns the name of the containing directory of the external file (including directories) identified by Name. (If more than one directory can contain Name, the directory name returned is implementation-defined.) The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if the external file does not have a containing directory.

```
function Extension (Name : in String) return String;
```

Returns the extension name corresponding to Name. The extension name is a portion of a simple name (not including any separator characters), typically used to identify the file class. If the external environment does not have extension names, then the null string is returned. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file.

```
function Base_Name (Name : in String) return String;
```

Returns the base name corresponding to Name. The base name is the remainder of a simple name after removing any extension and extension separators. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Compose (Containing_Directory : in String := "";  
                  Name : in String;  
                  Extension : in String := "") return String;
```

Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise Name is interpreted as a base name. The exception Name_Error is propagated if the string given as Containing_Directory is not null and does not allow the identification of a directory, or if the string given as Extension is not null and is not a possible extension, or if the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is non-null).

The following file and directory queries and types are provided:

```
type File_Kind is (Directory, Ordinary_File, Special_File);
```

The type File_Kind represents the kind of file represented by an external file or directory.

```
type File_Size is range 0 .. implementation-defined;
```

The type File_Size represents the size of an external file.

```
function Exists (Name : in String) return Boolean;
```

Returns True if external file represented by Name exists, and False otherwise. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Kind (Name : in String) return File_Kind;
```

Returns the kind of external file represented by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file.

```
function Size (Name : in String) return File_Size;
```

Returns the size of the external file represented by Name. The size of an external file is the number of stream elements contained in the file. If the external file is discontinuous (not all elements exist), the result is implementation-defined. If the external file is not an ordinary file, the result is implementation-defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

```
function Modification_Time (Name : in String) return Ada.Calendar.Time;
```

Returns the time that the external file represented by Name was most recently modified. If the external file is not an ordinary file, the result is implementation-defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Use_Error is propagated if the external environment does not support the reading the modification time of the file with the name given by Name (in the absence of Name_Error).

The following directory searching operations and types are provided:

```
type Directory_Entry_Type is limited private;
```

The type Directory_Entry_Type represents a single item in a directory. These items can only be created by the Get_Next_Entry procedure in this package. Information about the item can be obtained from the functions declared in this package. A default initialized object of this type is invalid; objects returned from Get_Next_Entry are valid.

```
type Filter_Type is array (File_Kind) of Boolean;
```

The type Filter_Type specifies which directory entries are provided from a search operation. If the Directory component is True, directory entries representing directories are provided. If the Ordinary_File component is True, directory entries representing ordinary files are provided. If the Special_File component is True, directory entries representing special files are provided.

```
type Search_Type is limited private;
```

The type Search_Type contains the state of a directory search. A default-initialized Search_Type object has no entries available (More_Entries returns False).

```
procedure Start_Search (Search : in out Search_Type;  
    Directory : in String;  
    Pattern : in String;  
    Filter : in Filter_Type := (others => True));
```

Starts a search in the directory entry in the directory named by Directory for entries matching Pattern. Pattern represents a file name matching pattern. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation-defined. Only items which match Filter will be returned. After a successful call on Start_Search, the object Search may have entries available, but it may have no entries available if no files or directories match Pattern and Filter. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error).

```
procedure End_Search (Search : in out Search_Type);
```

Ends the search represented by Search. After a successful call on End_Search, the object Search will have no entries available.

```
function More_Entries (Search : in Search_Type) return Boolean;
```

Returns True if more entries are available to be returned by a call to Get_Next_Entry for the specified search object, and False otherwise.

```
procedure Get_Next_Entry (Search : in out Search_Type;
```

```
Directory_Entry : out Directory_Entry_Type);
```

Returns the next `Directory_Entry` for the search described by `Search` that matches the pattern and filter. If no further matches are available, `Status_Error` is raised. It is implementation-defined as to whether the results returned by this routine are altered if the contents of the directory are altered while the `Search` object is valid (for example, by another program). The exception `Use_Error` is propagated if the external environment does not support continued searching of the directory represented by `Search`.

```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
    return String;
```

Returns the simple external name of the external file (including directories) represented by `Directory_Entry`. The format of the name returned is implementation-defined. The exception `Status_Error` is propagated if `Directory_Entry` is invalid.

```
function Full_Name (Directory_Entry : in Directory_Entry_Type) return String;
```

Returns the full external name of the external file (including directories) represented by `Directory_Entry`. The format of the name returned is implementation-defined. The exception `Status_Error` is propagated if `Directory_Entry` is invalid.

```
function Kind (Directory_Entry : in Directory_Entry_Type) return File_Kind;
```

Returns the kind of external file represented by `Directory_Entry`. The exception `Status_Error` is propagated if `Directory_Entry` is invalid.

```
function Size (Directory_Entry : in Directory_Entry_Type) return File_Size;
```

Returns the size of the external file represented by `Directory_Entry`. The size of an external file is the number of stream elements contained in the file. If the external file is discontinuous (not all elements exist), the result is implementation-defined. If the external file represented by `Directory_Entry` is not an ordinary file, the result is implementation-defined. The exception `Status_Error` is propagated if `Directory_Entry` is invalid. The exception `Constraint_Error` is propagated if the file size is not a value of type `File_Size`.

```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
    return Ada.Calendar.Time;
```

Returns the time that the external file represented by `Directory_Entry` was most recently modified. If the external file represented by `Directory_Entry` is not an ordinary file, the result is implementation-defined. The exception `Status_Error` is propagated if `Directory_Entry` is invalid. The exception `Use_Error` is propagated if the external environment does not support the reading the modification time of the file represented by `Directory_Entry`.

Implementation Requirements

For `Copy_File`, if `Source_Name` identifies an existing external ordinary file created by a predefined Ada Input-Output package, and `Target_Name` and `Form` can be used in the `Create` operation of that Input-Output package with mode `Out_File` without raising an exception, then `Copy_File` shall not propagate `Use_Error`.

Implementation Advice

If other information about a file is available (such as the owner or creation date) in a directory entry, the implementation should provide functions in a child package `Ada.Directories.Information` to retrieve it.

`Start_Search` should raise `Use_Error` if `Pattern` is malformed, but not if it could represent a file in the directory but does not actually do so.

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

NOTES

37 The file name operations `Containing_Directory`, `Full_Name`, `Simple_Name`, `Base_Name`, `Extension`, and `Compose` operate on file names, not external files. The files identified by these operations do not need to exist. `Name_Error` is raised only if the file name is malformed and cannot possibly identify a file.

38 Values of `Search_Type` and `Directory_Entry_Type` can be saved and queried later. However, another task or application can modify or delete the file represented by a `Directory_Entry_Type` value or the directory represented by a `Search_Type` value; such a value can only give the information valid at the time it is created. Therefore, long-term storage of these values is not recommended.

39 If the target system does not support directories inside of directories, `Is_Directory` will always return `False`, and `Containing_Directory` will always raise `Use_Error`.

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

Annex B: Interface to Other Languages

B.3 Interfacing with C

Replace paragraph 50: [AI95-00258-01]

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True.

by:

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.

Replace paragraph 60.2: [AI95-00216-01]

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

by:

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

B.3.1 The Package Interfaces.C.Strings

Replace paragraph 5: [AI95-00161-01]

```
type Chars_Ptr is private;
```

by:

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

Replace paragraph 6: [AI95-00276-01]

```
type chars_ptr_array is array (size_t range <>) of chars_ptr;
```

by:

```
type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;
```

Replace paragraph 50: [AI95-00242-01]

Equivalent to Update(Item, Offset, To_C(Str), Check).

by:

Equivalent to Update(Item, Offset, To_C(Str, Append_Nul => False), Check).

B.3.3 Pragma Unchecked_Union

Insert new clause: [AI95-00216-01]

A pragma Unchecked_Union specifies an interface correspondence between a given discriminated type and some C union. The pragma specifies that the associated type shall be given a representation that leaves no space for its discriminant(s).

Syntax

The form of a pragma Unchecked_Union is as follows:

pragma Unchecked_Union (*first_subtype_local_name*);

Legality Rules

Unchecked_Union is a representation pragma, specifying the unchecked union aspect of representation.

The *first_subtype_local_name* of a pragma Unchecked_Union shall denote an unconstrained discriminated record subtype having a *variant_part*.

A type to which a pragma Unchecked_Union applies is called an *unchecked union type*. A subtype of an unchecked union type is defined to be an *unchecked union subtype*. An object of an unchecked union type is defined to be an *unchecked union object*.

All component subtypes of an unchecked union type shall be C-compatible.

If a component subtype of an unchecked union type is subject to a per-object constraint, then the component subtype shall be an unchecked union subtype.

Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type.

A component declared in a *variant_part* of an unchecked union type shall not have a controlled, protected, or task part.

The completion of an incomplete or private type declaration having a *known_discriminant_part* shall not be an unchecked union type.

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

Static Semantics

An unchecked union type is eligible for convention C.

Discriminant_Check is suppressed for an unchecked union type.

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

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

Dynamic Semantics

A view of an unchecked union object (including a type conversion or function call) has *inferable discriminants* if it has a constrained nominal subtype, unless the object is a component of an enclosing unchecked union object that is subject to a per-object constraint and the enclosing object lacks inferable discriminants.

An expression of an unchecked union type has inferable discriminants if it is either a name of an object with inferable discriminants or a qualified expression whose *subtype_mark* denotes a constrained subtype.

Program_Error is raised in the following cases:

- Evaluation of the predefined equality operator for an unchecked union type if either of the operands lacks inferable discriminants.

- Evaluation of the predefined equality operator for a type which has a subcomponent of an unchecked union type whose nominal subtype is unconstrained.
- Evaluation of a membership test if the `subtype_mark` denotes a constrained unchecked union subtype and the expression lacks inferable discriminants.
- Conversion from a derived unchecked union type to an unconstrained non-unchecked-union type if the operand of the conversion lacks inferable discriminants.
- Execution of the default implementation of the Write or Read attribute of an unchecked union type.
- Execution of the default implementation of the Output or Input attribute of an unchecked union type if the type lacks default discriminant values.

Implementation Permissions

An implementation may require that `pragma Controlled` be specified for the type of an access subcomponent of an `Unchecked_Union` type.

NOTES

15 The use of an unchecked union to obtain the effect of an unchecked conversion results in erroneous execution (see 11.5). Execution of the following example is erroneous even if `Float'Size = Integer'Size`:

```

type T (Flag : Boolean := False) is
  record
    case Flag is
      when False =>
        F1 : Float := 0.0;
      when True =>
        F2 : Integer := 0;
    end case;
  end record;
pragma Unchecked_Union (T);

X : T;
Y : Integer := X.F2; -- erroneous

```

Annex C: Systems Programming

C.3.1 Protected Procedure Handlers

Replace paragraph 8: [AI95-00253-01]

The `Interrupt_Handler` pragma is only allowed immediately within a `protected_definition`. The corresponding `protected_type_declaration` shall be a library level declaration. In addition, any `object_declaration` of such a type shall be a library level declaration.

by:

The `Interrupt_Handler` pragma is only allowed immediately within a `protected_definition` where the corresponding subprogram is declared. The corresponding `protected_type_declaration` or `single_protected_declaration` shall be a library level declaration. In addition, any `object_declaration` of such a type shall be a library level declaration.

C.4 Preelaboration Requirements

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

- Any `subtype_mark` denotes a statically constrained subtype, with statically constrained subcomponents, if any;

the new paragraph:

- No `subtype_mark` denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type;

C.6 Shared Variable Control

Replace paragraph 7: [AI95-00272-01]

An *atomic* type is one to which a pragma `Atomic` applies. An *atomic* object (including a component) is one to which a pragma `Atomic` applies, or a component of an array to which a pragma `Atomic_Components` applies, or any object of an atomic type.

by:

An *atomic* type is one to which a pragma `Atomic` applies. An *atomic* object (including a component) is one to which a pragma `Atomic` applies, or a component of an array to which a pragma `Atomic_Components` applies, or any object of an atomic type, other than objects obtained by evaluating a slice.

Insert after paragraph 21: [AI95-00259-01]

If a pragma `Pack` applies to a type any of whose subcomponents are atomic, the implementation shall not pack the atomic subcomponents more tightly than that for which it can support indivisible reads and updates.

the new paragraphs:

Implementation Advice

A load or store of a volatile object whose size is a multiple of `System.Storage_Unit` and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others.

A load or store of an atomic object should, where possible, be implemented by a single load or store instruction.

Annex D: Real-Time Systems

D.2.2 Pragma Task_Dispatching_Policy

Replace paragraph 17: [AI95-00256-01]

Implementations are allowed to define other task dispatching policies, but need not support more than one such policy per partition.

by:

Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

D.3 Priority Ceiling Locking

Replace paragraph 15: [AI95-00256-01]

Implementations are allowed to define other locking policies, but need not support more than one such policy per partition.

by:

Implementations are allowed to define other locking policies, but need not support more than one locking policy per partition.

D.4 Entry Queuing Policies

Replace paragraph 15: [AI95-00256-01]

Implementations are allowed to define other queuing policies, but need not support more than one such policy per partition.

by:

Implementations are allowed to define other queuing policies, but need not support more than one queuing policy per partition.

D.7 Tasking Restrictions

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

No_Asynchronous_Control

There are no semantic dependences on the package Asynchronous_Task_Control.

the new paragraphs:

No_Calendar

There are no semantic dependencies on package Ada.Calendar.

No_Dynamic_Attachment

There is no call to any of the operations defined in package Ada.Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

No_Local_Protected_Objects

Protected objects shall be declared only at library level.

No_Protected_Type_Allocators

There are no allocators for protected types or types containing protected type components.

No_Relative_Delay

There are no `delay_relative_statements`.

No_Requeue_Statements

There are no `requeue_statements`.

No_Select_Statements

There are no `select_statements`.

No_Task_Attributes_Package

There are no semantic dependencies on package `Ada.Task_Attributes`.

Simple_Barriers

The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.

Replace paragraph 15: [AI95-00305-01]

This paragraph was deleted

by:

No_Task_Termination

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

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

Max_Tasks

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

the new paragraph:

Max_Entry_Queue_Length

`Max_Entry_Queue_Length` defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of `Program_Error` at the point of the call.

D.13 Run-time Profiles and the Ravenscar Profile

Insert new clause: [AI95-00249-01]

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

Syntax

The form of a pragma Profile is as follows:

pragma Profile (*profile_identifier* [*profile_argument_associations*]);

profile_argument_associations ::= *pragma_argument_association*, {*pragma_argument_association*}

Legality Rules

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

Static Semantics

A profile is equivalent to the set of configuration pragmas that is defined for each *profile_identifier*. The *profile_identifier* Ravenscar is equivalent to the following set of pragmas:

pragma Task_Dispatching_Policy (FIFO_Within_Priorities);

```

pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
    Max_Entry_Queue_Length => 1,
    Max_Protected_Entries => 1,
    Max_Task_Entries => 0,
    No_Abort_Statements,
    No_Asynchronous_Control,
    No_Calendar,
    No_Dynamic_Attachment,
    No_Dynamic_Priorities,
    No_Implicit_Heap_Allocations,
    No_Local_Protected_Objects,
    No_Protected_Type_Allocators,
    No_Relative_Delay,
    No_Requeue_Statements,
    No_Select_Statements,
    No_Task_Allocators,
    No_Task_Attributes_Package,
    No_Task_Hierarchy,
    No_Task_Termination,
    Simple_Barriers);

```

Post-Compilation Rules

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

NOTES

37 The effect of the Max_Entry_Queue_Length => 1 restriction applies only to protected entry queues due to the accompanying restriction of Max_Task_Entries => 0.

Annex E: Distributed Systems

E.2.2 Remote Types Library Units

Replace paragraph 8: [AI95-00240-01]

- if the full view of a type declared in the visible part of the library unit has a part that is of a non-remote access type, then that access type, or the type of some part that includes the access type subcomponent, shall have user-specified Read and Write attributes.

by:

- if the full view of a type declared in the visible part of the library unit has a part that is of a non-remote access type, then that access type, or the type of some part that includes the access type subcomponent, shall have Read and Write attributes specified by a visible `attribute_definition_clause`.

Replace paragraph 14: [AI95-00240-01]

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

by:

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall have either a nonlimited type or a type with available Read and Write attributes (see 13.13.2);

E.2.3 Remote Call Interface Library Units

Replace paragraph 14: [AI95-00240-01]

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

by:

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter, or a formal parameter of a limited type unless that limited type has available Read and Write attributes (see 13.13.2);

E.5 Partition Communication Subsystem

Replace paragraph 1: [AI95-00273-01]

The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package `System.RPC` is a language-defined interface to the PCS. An implementation conforming to this Annex shall use the RPC interface to implement remote subprogram calls.

by:

The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package `System.RPC` is a language-defined interface to the PCS.

Insert after paragraph 27: [AI95-00273-01]

A body for the package System.RPC need not be supplied by the implementation.

the new paragraph:

An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this clause.

Annex F: Information Systems

No changes in this section.

Annex G: Numerics

G.1.1 Complex Types

Replace paragraph 4: [AI95-00161-01]

```
type Imaginary is private;
```

by:

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

G.1.2 Complex Elementary Functions

Replace paragraph 15: [AI95-00185-01]

The real (resp., imaginary) component of the result of the Arcsin and Arccos (resp., Arctanh) functions is discontinuous as the parameter X crosses the real axis to the left of -1.0 or the right of 1.0 .

by:

The imaginary component of the result of the Arcsin, Arccos, and Arctanh functions is discontinuous as the parameter X crosses the real axis to the left of -1.0 or the right of 1.0 .

Replace paragraph 16: [AI95-00185-01]

The real (resp., imaginary) component of the result of the Arctan (resp., Arcsinh) function is discontinuous as the parameter X crosses the imaginary axis below $-i$ or above i .

by:

The real component of the result of the Arctan and Arcsinh functions is discontinuous as the parameter X crosses the imaginary axis below $-i$ or above i .

Replace paragraph 17: [AI95-00185-01]

The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis between $-i$ and i .

by:

The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis below $-i$ or above i .

Replace paragraph 20: [AI95-00185-01]

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

by:

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in Ada.Numerics.Generic_Elementary_Functions. (For Arctan and Arccot, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.)

G.2.2 Model-Oriented Attributes of Floating Point Types

Replace paragraph 3: [AI95-00256-01]

Yields the number of digits in the mantissa of the canonical form of the model numbers of T (see A.5.3). The value of this attribute shall be greater than or equal to $\text{Ceiling}(d * \log(10) / \log(T\text{Machine_Radix})) + 1$, where d is the requested decimal precision of T . In addition, it shall be less than or equal to the value of $T\text{Machine_Mantissa}$. This attribute yields a value of the type *universal_integer*.

by:

Yields the number of digits in the mantissa of the canonical form of the model numbers of T (see A.5.3). The value of this attribute shall be greater than or equal to

$$\text{ceiling}(d * \log(10) / \log(T\text{Machine_Radix})) + g$$

where d is the requested decimal precision of T , and g is 0 if Machine_Radix is a positive power of 10 and 1 otherwise. In addition, it shall be less than or equal to the value of $T\text{Machine_Mantissa}$. This attribute yields a value of the type *universal_integer*.

Annex H: Safety and Security

H.5 Pragma Detect_Blocking

Insert new clause: [AI95-00305-01]

The following pragma forces an implementation to detect potentially blocking operations within a protected operation.

Syntax

The form of a pragma Detect_Blocking is as follows:

pragma Detect_Blocking;

Dynamic Semantics

An implementation is required to detect a potentially blocking operation within a protected operation, and to raise Program_Error (see 9.5.1).

Post-Compilation Rules

A pragma Detect_Blocking is a configuration pragma.

Implementation Permissions

An implementation is allowed to reject a compilation_unit if a potentially blocking operation is present directly within an entry_body or the body of a protected subprogram.

NOTES

10 An operation that causes a task to be blocked within a foreign language domain is not defined to be potentially blocking, and need not be detected.

H.6 Pragma Partition_Elaboration_Policy

Insert new clause: [AI95-00265-01]

This clause defines a pragma for user control over elaboration policy.

Syntax

The form of a pragma Partition_Elaboration_Policy is as follows:

pragma Partition_Elaboration_Policy (*policy_identifier*);

The *policy_identifier* shall be either Sequential, Concurrent or an implementation-defined identifier.

Post-Compilation Rules

The pragma is a configuration pragma. It applies to all compilation units in a partition.

If the Sequential policy is specified for a partition then pragma Restrictions (No_Task_Hierarchy) shall also be specified for the partition.

Dynamic Semantics

Notwithstanding what this International Standard says elsewhere, this pragma allows partition elaboration rules concerning task activation and interrupt attachment to be changed. If the *policy_identifier* is Concurrent, or if there is no pragma Partition_Elaboration_Policy defined for the partition, then the rules defined elsewhere in this Standard apply.

If the partition elaboration policy is Sequential, all task activation for library-level tasks and all interrupt handler attachment for library-level interrupt handlers is deferred. The deferred task activation and handler attachment

occurs after the elaboration of all `library_items` prior to calling the main subprogram. At this point the Environment task is suspended until all deferred task activation and handler attachment is complete.

If any deferred task activation fails, `Tasking_Error` is raised in the Environment task. The Environment task and all tasks whose activations fail are terminated. If a number of dynamic interrupt handler attachments for the same interrupt are deferred then the most recent call of `Attach_Handler` or `Exchange_Handler` determines which handler is attached.

Implementation Advice

If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration then the partition is deadlocked and it is recommended that it be immediately terminated.

Implementation Permission

If the partition elaboration policy is Sequential and any task activation fails then an implementation may immediately terminate the active partition to mitigate the hazard posed by continuing to execute with a subset of the tasks being active.

Annex J: Obsolescent Features

J.10 Specific Suppression of Checks

Insert new clause: [AI95-00224-01]

Pragma Suppress can be used to suppress checks on specific entities.

Syntax

The form of a specific Suppress pragma is as follows:

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

Legality Rules

The **identifier** shall be the name of a check (see 11.5). The **name** shall statically denote some entity.

For a specific Suppress pragma that is immediately within a **package_specification**, the **name** shall denote an entity (or several overloaded subprograms) declared immediately within the **package_specification**.

Static Semantics

A specific Suppress pragma applies to the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a **package_specification**, to the end of the scope of the named entity. The pragma applies only to the named entity, or, for a subtype, on objects and values of its type. A specific Suppress pragma suppresses the named check for any entities to which it applies (see 11.5). Which checks are associated with a specific entity is not defined by this standard.

Implementation Permissions

An implementation is allowed to place restrictions on specific Suppress pragmas.

NOTES

- 3 An implementation may support an entity parameter on pragma Unsuppress (see 11.5).