Excerpt from

Ada 95 Reference Manual

Content:

Section 9, Section 13.1-13.9 Annex C och Annex D

Section 9: Tasks and Synchronization

The execution of an Ada program consists of the execution of one or more *tasks*. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it *interacts* with other tasks. The various forms of task interaction are described in this section, and include: 1

- the activation and termination of a task; 2
- a call on a protected subprogram of a *protected object*, providing exclusive read-write access, or concurrent read-only access to shared data; 3
- a call on an entry, either of another task, allowing for synchronous communication with that task, or of a protected object, allowing for asynchronous communication with one or more other tasks using that same protected object; 4
- a timed operation, including a simple delay statement, a timed entry call or accept, or a timed asynchronous select statement (see next item); 5
- an asynchronous transfer of control as part of an asynchronous select statement, where a task stops what it is doing and begins execution at a different point in response to the completion of an entry call or the expiration of a delay; 6
- an abort statement, allowing one task to cause the termination of another task. 7

In addition, tasks can communicate indirectly by reading and updating (unprotected) shared variables, presuming the access is properly synchronized through some other kind of task interaction. 8

Static Semantics

The properties of a task are defined by a corresponding task declaration and task_body, which together define a program unit called a *task unit*. 9

Dynamic Semantics

Over time, tasks proceed through various *states*. A task is initially *inactive*; upon activation, and prior to its *termination* it is either *blocked* (as part of some task interaction) or *ready* to run. While ready, a task competes for the available *execution resources* that it requires to run. 10

NOTES

1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way. 11

9.1 Task Units and Task Objects

A task unit is declared by a *task declaration*, which has a corresponding task_body. A task declaration may be a task_type_declaration, in which case it declares a named task type; alternatively, it may be a single_task_declaration, in which case it defines an anonymous task type, as well as declaring a named task object of that type. 1

Syntax

task_type_declaration ::= 2
  task type defining_identifier [known_discriminant_part] [**is** task_definition];

single_task_declaration ::= 3
  task defining_identifier [**is** task_definition];

4 task_definition ::=
    {task_item}
  [**private**
    {task_item}]
  end [*task*_identifier]

5 task_item ::= entry_declaration | representation_clause

6 task_body ::=
  task body defining_identifier **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*task*_identifier];

7 If a *task*_identifier appears at the end of a task_definition or task_body, it shall repeat the defining_identifier.

Legality Rules

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

Static Semantics

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

Dynamic Semantics

10 The elaboration of a task declaration elaborates the task_definition. The elaboration of a single_task_declaration also creates an object of an (anonymous) task type.

11 The elaboration of a task_definition creates the task type and its first subtype; it also includes the elaboration of the entry_declarations in the given order.

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

13 The elaboration of a task_body has no effect other than to establish that tasks of the type can from then on be activated without failing the Elaboration_Check.

14 The execution of a task_body is invoked by the activation of a task of the corresponding type (see 9.2).

15 The content of a task object of a given task type includes:

16 • The values of the discriminants of the task object, if any;

17 • An entry queue for each entry of the task object;

18 • A representation of the state of the associated task.

NOTES

19 2 Within the declaration or body of a task unit, the name of the task unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a subtype_mark).

20 3 The notation of a selected_component can be used to denote a discriminant of a task (see 4.1.3). Within a task unit, the name of a discriminant of the task type denotes the corresponding discriminant of the current instance of the unit.

21 4 A task type is a limited type (see 7.5), and hence has neither an assignment operation nor predefined equality operators. If an application needs to store and exchange task identities, it can do so by defining an access type designating the

corresponding task objects and by using access values for identification purposes. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7).

Examples

Examples of declarations of task types:

```
task type Server is
   entry Next_Work_Item(WI : in Work_Item);
   entry Shut_Down;
end Server;
```

```
task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;
```

Examples of declarations of single tasks:

```
task Controller is
   entry Request(Level)(D : Item);  -- a family of entries
end Controller;
```

```
task Parser is
   entry Next_Lexeme(L : in  Lexical_Element);
   entry Next_Action(A : out Parser_Action);
end;
```

```
task User;  -- has no entries
```

Examples of task objects:

```
Agent    : Server;
Teletype : Keyboard_Driver(TTY_ID);
Pool     : array(1 .. 10) of Keyboard_Driver;
```

Example of access type designating task objects:

```
type Keyboard is access Keyboard_Driver;
Terminal : Keyboard := new Keyboard_Driver(Term_ID);
```

9.2 Task Execution - Task Activation

Dynamic Semantics

The execution of a task of a given task type consists of the execution of the corresponding task_body. The initial part of this execution is called the *activation* of the task; it consists of the elaboration of the declarative_part of the task_body. Should an exception be propagated by the elaboration of its declarative_part, the activation of the task is defined to have *failed*, and it becomes a completed task.

A task object (which represents one task) can be created either as part of the elaboration of an object_declaration occurring immediately within some declarative region, or as part of the evaluation of an allocator. All tasks created by the elaboration of object_declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together. The activation of a task is associated with the innermost allocator or object_declaration that is responsible for its creation.

For tasks created by the elaboration of object_declarations of a given declarative region, the activations are initiated within the context of the handled_sequence_of_statements (and its associated exception_handlers if any — see 11.2), just prior to executing the statements of the _sequence. For a package without an explicit body or an explicit handled_sequence_of_statements, an implicit body or an implicit null_statement is assumed, as defined in 7.2.

4 For tasks created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, after completing any initialization for the object created by the allocator, and prior to returning the new access value.

5 The task that created the new tasks and initiated their activations (the *activator*) is blocked until all of these activations complete (successfully or not). Once all of these activations are complete, if the activation of any of the tasks has failed (due to the propagation of an exception), Tasking_Error is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any tasks that are aborted prior to completing their activation are ignored when determining whether to raise Tasking_Error.

6 Should the task that created the new tasks never reach the point where it would initiate the activations (due to an abort or the raising of an exception), the newly created tasks become terminated and are never activated.

NOTES

7 5 An entry of a task can be called before the task has been activated.

8 6 If several tasks are activated together, the execution of any of these tasks need not await the end of the activation of the other tasks.

9 7 A task can become completed during its activation either because of an exception or because it is aborted (see 9.8).

Examples

10 *Example of task activation:*

11
```
procedure P is
   A, B : Server;     -- elaborate the task objects A, B
   C    : Server;     -- elaborate the task object C
begin
   -- the tasks A, B, C are activated together before the first statement
   ...
end;
```

9.3 Task Dependence - Termination of Tasks

Dynamic Semantics

1 Each task (other than an environment task — see 10.2) *depends* on one or more masters (see 7.6.1), as follows:

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

3 • If the task is created by the elaboration of an object_declaration, it depends on each master that includes this elaboration.

4 Furthermore, if a task depends on a given master, it is defined to depend on the task that executes the master, and (recursively) on any master of that task.

5 A task is said to be *completed* when the execution of its corresponding task_body is completed. A task is said to be *terminated* when any finalization of the task_body has been performed (see 7.6.1). The first step of finalizing a master (including a task_body) is to wait for the termination of any tasks dependent on the master. The task executing the master is blocked until all the dependents have terminated. Any remaining finalization is then performed and the master is left.

Completion of a task (and the corresponding task_body) can occur when the task is blocked at a select_statement with an an open terminate_alternative (see 9.7.1); the open terminate_alternative is selected if and only if the following conditions are satisfied:

- The task depends on some completed master;
- Each task that depends on the master considered is either already terminated or similarly blocked at a select_statement with an open terminate_alternative.

When both conditions are satisfied, the task considered becomes completed, together with all tasks that depend on the master considered that are not yet completed.

NOTES

8 The full view of a limited private type can be a task type, or can have subcomponents of a task type. Creation of an object of such a type creates dependences according to the full type.

9 An object_renaming_declaration defines a new view of an existing entity and hence creates no further dependence.

10 The rules given for the collective completion of a group of tasks all blocked on select_statements with open terminate_alternatives ensure that the collective completion can occur only when there are no remaining active tasks that could call one of the tasks being collectively completed.

11 If two or more tasks are blocked on select_statements with open terminate_alternatives, and become completed collectively, their finalization actions proceed concurrently.

12 The completion of a task can occur due to any of the following:

- the raising of an exception during the elaboration of the declarative_part of the corresponding task_body;
- the completion of the handled_sequence_of_statements of the corresponding task_body;
- the selection of an open terminate_alternative of a select_statement in the corresponding task_body;
- the abort of the task.

Examples

Example of task dependence:

```
declare
   type Global is access Server;         -- see 9.1
   A, B : Server;
   G    : Global;
begin
   -- activation of A and B
   declare
      type Local is access Server;
      X : Global := new Server;  -- activation of X.all
      L : Local  := new Server;  -- activation of L.all
      C : Server;
   begin
      -- activation of C
      G := X;  -- both G and X designate the same task object
      ...
   end;  -- await termination of C and L.all (but not X.all)
   ...
end;  -- await termination of A, B, and G.all
```

9.4 Protected Units and Protected Objects

A *protected object* provides coordinated access to shared data, through calls on its visible *protected operations*, which can be *protected subprograms* or *protected entries*. A *protected unit* is declared by a *protected declaration*, which has a corresponding protected_body. A protected declaration may be a protected_type_declaration, in which case it declares a named protected type; alternatively, it may be a single_protected_declaration, in which case it defines an anonymous protected type, as well as declaring a named protected object of that type.

Syntax

2 protected_type_declaration ::=
protected type defining_identifier [known_discriminant_part] **is** protected_definition;

3 single_protected_declaration ::=
protected defining_identifier **is** protected_definition;

4 protected_definition ::=
{ protected_operation_declaration }
[**private**
{ protected_element_declaration }]
end [*protected*_identifier]

5 protected_operation_declaration ::= subprogram_declaration
| entry_declaration
| representation_clause

6 protected_element_declaration ::= protected_operation_declaration
| component_declaration

7 protected_body ::=
protected body defining_identifier **is**
{ protected_operation_item }
end [*protected*_identifier];

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

9 If a *protected*_identifier appears at the end of a protected_definition or protected_body, it shall repeat the defining_identifier.

Legality Rules

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

Static Semantics

11 A protected_definition defines a protected type and its first subtype. The list of protected_operation_declarations of a protected_definition, together with the known_discriminant_part, if any, is called the visible part of the protected unit. The optional list of protected_element_declarations after the reserved word **private** is called the private part of the protected unit.

Dynamic Semantics

12 The elaboration of a protected declaration elaborates the protected_definition. The elaboration of a single_protected_declaration also creates an object of an (anonymous) protected type.

13 The elaboration of a protected_definition creates the protected type and its first subtype; it also includes the elaboration of the component_declarations and protected_operation_declarations in the given order.

14 As part of the initialization of a protected object, any per-object constraints (see 3.8) are elaborated.

15 The elaboration of a protected_body has no other effect than to establish that protected operations of the type can from then on be called without failing the Elaboration_Check.

16 The content of an object of a given protected type includes:

17 • The values of the components of the protected object, including (implicitly) an entry queue for each entry declared for the protected object;

- A representation of the state of the execution resource *associated* with the protected object (one such resource is associated with each protected object). 18

The execution resource associated with a protected object has to be acquired to read or update any components of the protected object; it can be acquired (as part of a protected action — see 9.5.1) either for concurrent read-only access, or for exclusive read-write access. 19

As the first step of the *finalization* of a protected object, each call remaining on any entry queue of the object is removed from its queue and Program_Error is raised at the place of the corresponding entry_call_statement. 20

NOTES

13 Within the declaration or body of a protected unit, the name of the protected unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a subtype_mark). 21

14 A selected_component can be used to denote a discriminant of a protected object (see 4.1.3). Within a protected unit, the name of a discriminant of the protected type denotes the corresponding discriminant of the current instance of the unit. 22

15 A protected type is a limited type (see 7.5), and hence has neither an assignment operation nor predefined equality operators. 23

16 The bodies of the protected operations given in the protected_body define the actions that take place upon calls to the protected operations. 24

17 The declarations in the private part are only visible within the private part and the body of the protected unit. 25

Examples

Example of declaration of protected type and corresponding body: 26

```
protected type Resource is                                          27
   entry Seize;
   procedure Release;
private
   Busy : Boolean := False;
end Resource;

protected body Resource is                                          28
   entry Seize when not Busy is
   begin
      Busy := True;
   end Seize;

   procedure Release is                                             29
   begin
      Busy := False;
   end Release;
end Resource;
```

Example of a single protected declaration and corresponding body: 30

```
protected Shared_Array is                                           31
   -- Index, Item, and Item_Array are global types
   function  Component    (N : in Index) return Item;
   procedure Set_Component(N : in Index; E : in  Item);
private
   Table : Item_Array(Index) := (others => Null_Item);
end Shared_Array;

protected body Shared_Array is                                      32
   function Component(N : in Index) return Item is
   begin
      return Table(N);
   end Component;
```

```
      procedure Set_Component(N : in Index; E : in Item) is
      begin
         Table(N) := E;
      end Set_Component;
   end Shared_Array;
```

Examples of protected objects:

```
   Control  : Resource;
   Flags    : array(1 .. 100) of Resource;
```

9.5 Intertask Communication

The primary means for intertask communication is provided by calls on entries and protected subprograms. Calls on protected subprograms allow coordinated access to shared data objects. Entry calls allow for blocking the caller until a given condition is satisfied (namely, that the corresponding entry is open — see 9.5.3), and then communicating data or control information directly with another task or indirectly via a shared protected object.

Static Semantics

Any call on an entry or on a protected subprogram identifies a *target object* for the operation, which is either a task (for an entry call) or a protected object (for an entry call or a protected subprogram call). The target object is considered an implicit parameter to the operation, and is determined by the operation name (or prefix) used in the call on the operation, as follows:

- If it is a direct_name or expanded name that denotes the declaration (or body) of the operation, then the target object is implicitly specified to be the current instance of the task or protected unit immediately enclosing the operation; such a call is defined to be an *internal call*;
- If it is a selected_component that is not an expanded name, then the target object is explicitly specified to be the task or protected object denoted by the prefix of the name; such a call is defined to be an *external call*;
- If the name or prefix is a dereference (implicit or explicit) of an access-to-protected-subprogram value, then the target object is determined by the prefix of the Access attribute_reference that produced the access value originally, and the call is defined to be an *external call*;
- If the name or prefix denotes a subprogram_renaming_declaration, then the target object is as determined by the name of the renamed entity.

A corresponding definition of target object applies to a requeue_statement (see 9.5.4), with a corresponding distinction between an *internal requeue* and an *external requeue*.

Dynamic Semantics

Within the body of a protected operation, the current instance (see 8.6) of the immediately enclosing protected unit is determined by the target object specified (implicitly or explicitly) in the call (or requeue) on the protected operation.

Any call on a protected procedure or entry of a target protected object is defined to be an update to the object, as is a requeue on such an entry.

9.5.1 Protected Subprograms and Protected Actions

A *protected subprogram* is a subprogram declared immediately within a protected_definition. Protected procedures provide exclusive read-write access to the data of a protected object; protected functions provide concurrent read-only access to the data.

Static Semantics

Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable (updating is permitted).

Dynamic Semantics

For the execution of a call on a protected subprogram, the evaluation of the name or prefix and of the parameter associations, and any assigning back of **in out** or **out** parameters, proceeds as for a normal subprogram call (see 6.4). If the call is an internal call (see 9.5), the body of the subprogram is executed as for a normal subprogram call. If the call is an external call, then the body of the subprogram is executed as part of a new *protected action* on the target protected object; the protected action completes after the body of the subprogram is executed. A protected action can also be started by an entry call (see 9.5.3).

A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a protected function. This rule is expressible in terms of the execution resource associated with the protected object:

- *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either for concurrent read-only access if the protected action is for a call on a protected function, or for exclusive read-write access otherwise;
- *Completing* the protected action corresponds to *releasing* the associated execution resource.

After performing an operation on a protected object other than a call on a protected function, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3).

Bounded (Run-Time) Errors

During a protected action, it is a bounded error to invoke an operation that is *potentially blocking*. The following are defined to be potentially blocking operations:

- a select_statement;
- an accept_statement;
- an entry_call_statement;
- a delay_statement;
- an abort_statement;
- task creation or activation;
- an external call on a protected subprogram (or an external requeue) with the same target object as that of the protected action;
- a call on a subprogram whose body contains a potentially blocking operation.

17 If the bounded error is detected, Program_Error is raised. If not detected, the bounded error might result in deadlock or a (nested) protected action on the same target object.

18 Certain language-defined subprograms are potentially blocking. In particular, the subprograms of the language-defined input-output packages that manipulate files (implicitly or explicitly) are potentially blocking. Other potentially blocking subprograms are identified where they are defined. When not specified as potentially blocking, a language-defined subprogram is nonblocking.

NOTES

19 18 If two tasks both try to start a protected action on a protected object, and at most one is calling a protected function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it might be consuming processing resources while it awaits its turn. There is no language-defined ordering or queuing presumed for tasks competing to start a protected action — on a multiprocessor such tasks might use busy-waiting; for monoprocessor considerations, see D.3, ‘‘Priority Ceiling Locking’’.

20 19 The body of a protected unit may contain declarations and bodies for local subprograms. These are not visible outside the protected unit.

21 20 The body of a protected function can contain internal calls on other protected functions, but not protected procedures, because the current instance is a constant. On the other hand, the body of a protected procedure can contain internal calls on both protected functions and procedures.

22 21 From within a protected action, an internal call on a protected subprogram, or an external call on a protected subprogram with a different target object is not considered a potentially blocking operation.

Examples

23 *Examples of protected subprogram calls (see 9.4):*

24
```
Shared_Array.Set_Component(N, E);
E := Shared_Array.Component(M);
Control.Release;
```

9.5.2 Entries and Accept Statements

1 Entry_declarations, with the corresponding entry_bodies or accept_statements, are used to define potentially queued operations on tasks and protected objects.

Syntax

2 entry_declaration ::=
entry defining_identifier [(discrete_subtype_definition)] parameter_profile;

3 accept_statement ::=
accept *entry*_direct_name [(entry_index)] parameter_profile [**do**
handled_sequence_of_statements
end [*entry*_identifier]];

4 entry_index ::= expression

5 entry_body ::=
entry defining_identifier entry_body_formal_part entry_barrier **is**
declarative_part
begin
handled_sequence_of_statements
end [*entry*_identifier];

6 entry_body_formal_part ::= [(entry_index_specification)] parameter_profile

7 entry_barrier ::= **when** condition

8 entry_index_specification ::= **for** defining_identifier **in** discrete_subtype_definition

9 If an *entry*_identifier appears at the end of an accept_statement, it shall repeat the *entry*_direct_name. If an *entry*_identifier appears at the end of an entry_body, it shall repeat the defining_identifier.

An entry_declaration is allowed only in a protected or task declaration. 10

Name Resolution Rules

In an accept_statement, the expected profile for the *entry*_direct_name is that of the entry_declaration; the expected type for an entry_index is that of the subtype defined by the discrete_subtype_definition of the corresponding entry_declaration. 11

Within the handled_sequence_of_statements of an accept_statement, if a selected_component has a prefix that denotes the corresponding entry_declaration, then the entity denoted by the prefix is the accept_statement, and the selected_component is interpreted as an expanded name (see 4.1.3); the selector_name of the selected_component has to be the identifier for some formal parameter of the accept_statement. 12

Legality Rules

An entry_declaration in a task declaration shall not contain a specification for an access parameter (see 3.10). 13

For an accept_statement, the innermost enclosing body shall be a task_body, and the *entry*_direct_name shall denote an entry_declaration in the corresponding task declaration; the profile of the accept_statement shall conform fully to that of the corresponding entry_declaration. An accept_statement shall have a parenthesized entry_index if and only if the corresponding entry_declaration has a discrete_subtype_definition. 14

An accept_statement shall not be within another accept_statement that corresponds to the same entry_declaration, nor within an asynchronous_select inner to the enclosing task_body. 15

An entry_declaration of a protected unit requires a completion, which shall be an entry_body, and every entry_body shall be the completion of an entry_declaration of a protected unit. The profile of the entry_body shall conform fully to that of the corresponding declaration. 16

An entry_body_formal_part shall have an entry_index_specification if and only if the corresponding entry_declaration has a discrete_subtype_definition. In this case, the discrete_subtype_definitions of the entry_declaration and the entry_index_specification shall fully conform to one another (see 6.3.1). 17

A name that denotes a formal parameter of an entry_body is not allowed within the entry_barrier of the entry_body. 18

Static Semantics

The parameter modes defined for parameters in the parameter_profile of an entry_declaration are the same as for a subprogram_declaration and have the same meaning (see 6.2). 19

An entry_declaration with a discrete_subtype_definition (see 3.6) declares a *family* of distinct entries having the same profile, with one such entry for each value of the *entry index subtype* defined by the discrete_subtype_definition. A name for an entry of a family takes the form of an indexed_component, where the prefix denotes the entry_declaration for the family, and the index value identifies the entry within the family. The term *single entry* is used to refer to any entry other than an entry of an entry family. 20

In the entry_body for an entry family, the entry_index_specification declares a named constant whose subtype is the entry index subtype defined by the corresponding entry_declaration; the value of the *named entry index* identifies which entry of the family was called. 21

Dynamic Semantics

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

23 The actions to be performed when an entry is called are specified by the corresponding accept_statements (if any) for an entry of a task unit, and by the corresponding entry_body for an entry of a protected unit.

24 For the execution of an accept_statement, the entry_index, if any, is first evaluated and converted to the entry index subtype; this index value identifies which entry of the family is to be accepted. Further execution of the accept_statement is then blocked until a caller of the corresponding entry is selected (see 9.5.3), whereupon the handled_sequence_of_statements, if any, of the accept_statement is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call. Upon completion of the handled_sequence_of_statements, the accept_statement completes and is left. When an exception is propagated from the handled_sequence_of_statements of an accept_statement, the same exception is also raised by the execution of the corresponding entry_call_statement.

25 The above interaction between a calling task and an accepting task is called a *rendezvous*. After a rendezvous, the two tasks continue their execution independently.

26 An entry_body is executed when the condition of the entry_barrier evaluates to True and a caller of the corresponding single entry, or entry of the corresponding entry family, has been selected (see 9.5.3). For the execution of the entry_body, the declarative_part of the entry_body is elaborated, and the handled_sequence_of_statements of the body is executed, as for the execution of a subprogram_body. The value of the named entry index, if any, is determined by the value of the entry index specified in the *entry*_name of the selected entry call (or intermediate requeue_statement — see 9.5.4).

NOTES

27 22 A task entry has corresponding accept_statements (zero or more), whereas a protected entry has a corresponding entry_body (exactly one).

28 23 A consequence of the rule regarding the allowed placements of accept_statements is that a task can execute accept_statements only for its own entries.

29 24 A return_statement (see 6.5) or a requeue_statement (see 9.5.4) may be used to complete the execution of an accept_statement or an entry_body.

30 25 The condition in the entry_barrier may reference anything visible except the formal parameters of the entry. This includes the entry index (if any), the components (including discriminants) of the protected object, the Count attribute of an entry of that protected object, and data global to the protected unit.

31 The restriction against referencing the formal parameters within an entry_barrier ensures that all calls of the same entry see the same barrier value. If it is necessary to look at the parameters of an entry call before deciding whether to handle it, the entry_barrier can be ‘‘**when** True’’ and the caller can be requeued (on some private entry) when its parameters indicate that it cannot be handled immediately.

Examples

32 *Examples of entry declarations:*

33
```
entry Read(V : out Item);
entry Seize;
entry Request(Level)(D : Item);  -- a family of entries
```

Examples of accept statements:

```
accept Shut_Down;
accept Read(V : out Item) do
   V := Local_Item;
end Read;
accept Request(Low)(D : Item) do
   ...
end Request;
```

9.5.3 Entry Calls

An entry_call_statement (an *entry call*) can appear in various contexts. A *simple* entry call is a stand-alone statement that represents an unconditional call on an entry of a target task or a protected object. Entry calls can also appear as part of select_statements (see 9.7).

Syntax

entry_call_statement ::= *entry*_name [actual_parameter_part];

Name Resolution Rules

The *entry*_name given in an entry_call_statement shall resolve to denote an entry. The rules for parameter associations are the same as for subprogram calls (see 6.4 and 6.4.1).

Static Semantics

The *entry*_name of an entry_call_statement specifies (explicitly or implicitly) the target object of the call, the entry or entry family, and the entry index, if any (see 9.5).

Dynamic Semantics

Under certain circumstances (detailed below), an entry of a task or protected object is checked to see whether it is *open* or *closed*:

- An entry of a task is open if the task is blocked on an accept_statement that corresponds to the entry (see 9.5.2), or on a selective_accept (see 9.7.1) with an open accept_alternative that corresponds to the entry; otherwise it is closed.
- An entry of a protected object is open if the condition of the entry_barrier of the corresponding entry_body evaluates to True; otherwise it is closed. If the evaluation of the condition propagates an exception, the exception Program_Error is propagated to all current callers of all entries of the protected object.

For the execution of an entry_call_statement, evaluation of the name and of the parameter associations is as for a subprogram call (see 6.4). The entry call is then *issued*: For a call on an entry of a protected object, a new protected action is started on the object (see 9.5.1). The named entry is checked to see if it is open; if open, the entry call is said to be *selected immediately*, and the execution of the call proceeds as follows:

- For a call on an open entry of a task, the accepting task becomes ready and continues the execution of the corresponding accept_statement (see 9.5.2).
- For a call on an open entry of a protected object, the corresponding entry_body is executed (see 9.5.2) as part of the protected action.

If the accept_statement or entry_body completes other than by a requeue (see 9.5.4), return is made to the caller (after servicing the entry queues — see below); any necessary assigning back of formal to actual parameters occurs, as for a subprogram call (see 6.4.1); such assignments take place outside of any protected action.

12 If the named entry is closed, the entry call is added to an *entry queue* (as part of the protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; there is a separate (logical) entry queue for each entry of a given task or protected object (including each entry of an entry family).

13 When a queued call is *selected*, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called *servicing* the entry queue. An entry with queued calls can be serviced under the following circumstances:

14 • When the associated task reaches a corresponding accept_statement, or a selective_accept with a corresponding open accept_alternative;

15 • If after performing, as part of a protected action on the associated protected object, an operation on the object other than a call on a protected function, the entry is checked and found to be open.

16 If there is at least one call on a queue corresponding to an open entry, then one such call is selected according to the *entry queuing policy* in effect (see below), and the corresponding accept_statement or entry_body is executed as above for an entry call that is selected immediately.

17 The entry queuing policy controls selection among queued calls both for task and protected entry queues. The default entry queuing policy is to select calls on a given entry queue in order of arrival. If calls from two or more queues are simultaneously eligible for selection, the default entry queuing policy does not specify which queue is serviced first. Other entry queuing policies can be specified by pragmas (see D.4).

18 For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes.

19 For an entry call that is added to a queue, and that is not the triggering_statement of an asynchronous_select (see 9.7.4), the calling task is blocked until the call is cancelled, or the call is selected and a corresponding accept_statement or entry_body completes without requeuing. In addition, the calling task is blocked during a rendezvous.

20 An attempt can be made to cancel an entry call upon an abort (see 9.8) and as part of certain forms of select_statement (see 9.7.2, 9.7.3, and 9.7.4). The cancellation does not take place until a point (if any) when the call is on some entry queue, and not protected from cancellation as part of a requeue (see 9.5.4); at such a point, the call is removed from the entry queue and the call completes due to the cancellation. The cancellation of a call on an entry of a protected object is a protected action, and as such cannot take place while any other protected action is occurring on the protected object. Like any protected action, it includes servicing of the entry queues (in case some entry barrier depends on a Count attribute).

21 A call on an entry of a task that has already completed its execution raises the exception Tasking_Error at the point of the call; similarly, this exception is raised at the point of the call if the called task completes its execution or becomes abnormal before accepting the call or completing the rendezvous (see 9.8). This applies equally to a simple entry call and to an entry call as part of a select_statement.

Implementation Permissions

22 An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete.

When the entry of a protected object is checked to see whether it is open, the implementation need not reevaluate the condition of the corresponding entry_barrier if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution (or cancellation) of a protected procedure or entry call on the object since the condition was last evaluated.

An implementation may evaluate the conditions of all entry_barriers of a given protected object any time any entry of the object is checked to see if it is open.

When an attempt is made to cancel an entry call, the implementation need not make the attempt using the thread of control of the task (or interrupt) that initiated the cancellation; in particular, it may use the thread of control of the caller itself to attempt the cancellation, even if this might allow the entry call to be selected in the interim.

NOTES

26 If an exception is raised during the execution of an entry_body, it is propagated to the corresponding caller (see 11.4).

27 For a call on a protected entry, the entry is checked to see if it is open prior to queuing the call, and again thereafter if its Count attribute (see 9.9) is referenced in some entry barrier.

28 In addition to simple entry calls, the language permits timed, conditional, and asynchronous entry calls (see 9.7.2, 9.7.3, and see 9.7.4).

29 The condition of an entry_barrier is allowed to be evaluated by an implementation more often than strictly necessary, even if the evaluation might have side effects. On the other hand, an implementation need not reevaluate the condition if nothing it references was updated by an intervening protected action on the protected object, even if the condition references some global variable that might have been updated by an action performed from outside of a protected action.

Examples

Examples of entry calls:

```
Agent.Shut_Down;                      -- see 9.1
Parser.Next_Lexeme(E);                -- see 9.1
Pool(5).Read(Next_Char);              -- see 9.1
Controller.Request(Low)(Some_Item);   -- see 9.1
Flags(3).Seize;                       -- see 9.4
```

9.5.4 Requeue Statements

A requeue_statement can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue. Such a *requeue* can be performed with or without allowing an intermediate cancellation of the call, due to an abort or the expiration of a delay.

Syntax

requeue_statement ::= **requeue** *entry*_name [**with abort**];

Name Resolution Rules

The *entry*_name of a requeue_statement shall resolve to denote an entry (the *target entry*) that either has no parameters, or that has a profile that is type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.

Legality Rules

A requeue_statement shall be within a callable construct that is either an entry_body or an accept_statement, and this construct shall be the innermost enclosing body or callable construct.

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

6 In a requeue_statement of an accept_statement of some task unit, either the target object shall be a part of a formal parameter of the accept_statement, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing accept_statement of the task unit. In a requeue_statement of an entry_body of some protected unit, either the target object shall be a part of a formal parameter of the entry_body, or the accessibility level of the target object shall not be statically deeper than that of the entry_declaration.

Dynamic Semantics

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

8 For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3).

9 For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct:

10 • if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object — see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);

11 • if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object — see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3).

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

13 If the requeue_statement includes the reserved words **with abort** (it is a *requeue-with-abort*), then:

14 • if the original entry call has been aborted (see 9.8), then the requeue acts as an abort completion point for the call, and the call is cancelled and no requeue is performed;

15 • if the original entry call was timed (or conditional), then the original expiration time is the expiration time for the requeued call.

16 If the reserved words **with abort** do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement.

NOTES

17 30 A requeue is permitted from a single entry to an entry of an entry family, or vice-versa. The entry index, if any, plays no part in the subtype conformance check between the profiles of the two entries; an entry index is part of the *entry*_name for an entry of a family.

Examples

18 *Examples of requeue statements:*

19
```
requeue Request(Medium) with abort;
                        -- requeue on a member of an entry family of the current task, see 9.1
```

20
```
requeue Flags(I).Seize;
                        -- requeue on an entry of an array component, see 9.4
```

9.6 Delay Statements, Duration, and Time

A delay_statement is used to block further execution until a specified *expiration time* is reached. The expiration time can be specified either as a particular point in time (in a delay_until_statement), or in seconds from the current time (in a delay_relative_statement). The language-defined package Calendar provides definitions for a type Time and associated operations, including a function Clock that returns the current time.

Syntax

delay_statement ::= delay_until_statement | delay_relative_statement

delay_until_statement ::= **delay until** *delay*_expression;

delay_relative_statement ::= **delay** *delay*_expression;

Name Resolution Rules

The expected type for the *delay*_expression in a delay_relative_statement is the predefined type Duration. The *delay*_expression in a delay_until_statement is expected to be of any nonlimited type.

Legality Rules

There can be multiple time bases, each with a corresponding clock, and a corresponding *time type*. The type of the *delay*_expression in a delay_until_statement shall be a time type — either the type Time defined in the language-defined package Calendar (see below), or some other implementation-defined time type (see D.8).

Static Semantics

There is a predefined fixed point type named Duration, declared in the visible part of package Standard; a value of type Duration is used to represent the length of an interval of time, expressed in seconds. The type Duration is not specific to a particular time base, but can be used with any time base.

A value of the type Time in package Calendar, or of some other implementation-defined time type, represents a time as reported by a corresponding clock.

The following language-defined library package exists:

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

  subtype Year_Number  is Integer range 1901 ..  2099;
  subtype Month_Number is Integer range 1 ..  12;
  subtype Day_Number   is Integer range 1 ..  31;
  subtype Day_Duration is Duration range 0.0 ..  86_400.0;

  function Clock return Time;

  function Year   (Date : Time) return Year_Number;
  function Month  (Date : Time) return Month_Number;
  function Day    (Date : Time) return Day_Number;
  function Seconds(Date : Time) return Day_Duration;

  procedure Split (Date  : in Time;
                   Year    : out Year_Number;
                   Month   : out Month_Number;
                   Day     : out Day_Number;
                   Seconds : out Day_Duration);

  function Time_Of(Year  : Year_Number;
                   Month   : Month_Number;
                   Day     : Day_Number;
                   Seconds : Day_Duration := 0.0)
   return Time;
```

```
16        function "+" (Left : Time;   Right : Duration) return Time;
          function "+" (Left : Duration; Right : Time) return Time;
          function "-" (Left : Time;   Right : Duration) return Time;
          function "-" (Left : Time;   Right : Time) return Duration;
17        function "<" (Left, Right : Time) return Boolean;
          function "<="(Left, Right : Time) return Boolean;
          function ">" (Left, Right : Time) return Boolean;
          function ">="(Left, Right : Time) return Boolean;
18        Time_Error : exception;
19     private
          ... -- not specified by the language
       end Ada.Calendar;
```

Dynamic Semantics

20 For the execution of a delay_statement, the *delay*_expression is first evaluated. For a delay_until_ statement, the expiration time for the delay is the value of the *delay*_expression, in the time base associated with the type of the expression. For a delay_relative_statement, the expiration time is defined as the current time, in the time base associated with relative delays, plus the value of the *delay*_expression converted to the type Duration, and then rounded up to the next clock tick. The time base associated with relative delays is as defined in D.9, ‘‘Delay Accuracy’’ or is implementation defined.

21 The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes ready again. If the expiration time has already passed, the task is not blocked.

22 If an attempt is made to *cancel* the delay_statement (as part of an asynchronous_select or abort — see 9.7.4 and 9.8), the _statement is cancelled if the expiration time has not yet passed, thereby completing the delay_statement.

23 The time base associated with the type Time of package Calendar is implementation defined. The function Clock of package Calendar returns a value representing the current time for this time base. The implementation-defined value of the named number System.Tick (see 13.7) is an approximation of the length of the real-time interval during which the value of Calendar.Clock remains constant.

24 The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined timezone; the procedure Split returns all four corresponding values. Conversely, the function Time_Of combines a year number, a month number, a day number, and a duration, into a value of type Time. The operators "+" and "–" for addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning.

25 If Time_Of is called with a seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a seconds value of 0.0. The value returned by the function Seconds or through the Seconds parameter of the procedure Split is always less than 86_400.0.

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

Implementation Requirements

27 The implementation of the type Duration shall allow representation of time intervals (both positive and negative) up to at least 86400 seconds (one day); Duration’Small shall not be greater than twenty milliseconds. The implementation of the type Time shall allow representation of all dates with year numbers in the range of Year_Number; it may allow representation of other dates as well (both earlier and later).

Implementation Permissions

An implementation may define additional time types (see D.8). 28

An implementation may raise Time_Error if the value of a *delay*_expression in a delay_until_statement of a select_statement represents a time more than 90 days past the current time. The actual limit, if any, is implementation-defined. 29

Implementation Advice

Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds. 30

The time base for delay_relative_statements should be monotonic; it need not be the same time base as used for Calendar.Clock. 31

NOTES

31 A delay_relative_statement with a negative value of the *delay*_expression is equivalent to one with a zero value. 32

32 A delay_statement may be executed by the environment task; consequently delay_statements may be executed as part of the elaboration of a library_item or the execution of the main subprogram. Such statements delay the environment task (see 10.2). 33

33 A delay_statement is an abort completion point and a potentially blocking operation, even if the task is not actually blocked. 34

34 There is no necessary relationship between System.Tick (the resolution of the clock of package Calendar) and Duration'Small (the *small* of type Duration). 35

35 Additional requirements associated with delay_statements are given in D.9, ''Delay Accuracy''. 36

Examples

Example of a relative delay statement: 37

```
delay 3.0;   -- delay 3.0 seconds
```
38

Example of a periodic task: 39

```
declare
   use Ada.Calendar;
   Next_Time : Time := Clock + Period;
                       -- Period is a global constant of type Duration
begin
   loop                -- repeated every Period seconds
      delay until Next_Time;
      ... -- perform some actions
      Next_Time := Next_Time + Period;
   end loop;
end;
```
40

9.7 Select Statements

There are four forms of the select_statement. One form provides a selective wait for one or more select_alternatives. Two provide timed and conditional entry calls. The fourth provides asynchronous transfer of control. 1

Syntax

```
select_statement ::=
  selective_accept
 | timed_entry_call
 | conditional_entry_call
 | asynchronous_select
```
2

Examples

3 *Example of a select statement:*

4
```
select
   accept Driver_Awake_Signal;
or
   delay 30.0*Seconds;
   Stop_The_Train;
end select;
```

9.7.1 Selective Accept

1 This form of the select_statement allows a combination of waiting for, and selecting from, one or more alternatives. The selection may depend on conditions associated with each alternative of the selective_accept.

Syntax

2
```
selective_accept ::=
  select
   [guard]
    select_alternative
{ or
   [guard]
    select_alternative }
[ else
   sequence_of_statements ]
  end select;
```

3 guard ::= **when** condition =>

4
```
select_alternative ::=
   accept_alternative
  | delay_alternative
  | terminate_alternative
```

5
```
accept_alternative ::=
  accept_statement [sequence_of_statements]
```

6
```
delay_alternative ::=
  delay_statement [sequence_of_statements]
```

7 terminate_alternative ::= **terminate**;

8 A selective_accept shall contain at least one accept_alternative. In addition, it can contain:

9 • a terminate_alternative (only one); or

10 • one or more delay_alternatives; or

11 • an *else part* (the reserved word **else** followed by a sequence_of_statements).

12 These three possibilities are mutually exclusive.

Legality Rules

13 If a selective_accept contains more than one delay_alternative, then all shall be delay_relative_statements, or all shall be delay_until_statements for the same time type.

Dynamic Semantics

14 A select_alternative is said to be *open* if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be *closed* otherwise.

For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. For an open delay_alternative, the *delay*_expression is also evaluated. Similarly, for an open accept_alternative for an entry of a family, the entry_index is also evaluated. These evaluations are performed in an arbitrary order, except that a *delay*_expression or entry_index is not evaluated until after evaluating the corresponding condition, if any. Selection and execution of one open alternative, or of the else part, then completes the execution of the selective_accept; the rules for this selection are described below. 15

Open accept_alternatives are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected according to the entry queuing policy in effect (see 9.5.3 and D.4). When such an alternative is selected, the selected call is removed from its entry queue and the handled_sequence_of_statements (if any) of the corresponding accept_statement is executed; after the rendezvous completes any subsequent sequence_of_statements of the alternative is executed. If no selection is immediately possible (in the above sense) and there is no else part, the task blocks until an open alternative can be selected. 16

Selection of the other forms of alternative or of an else part is performed as follows: 17

- An open delay_alternative is selected when its expiration time is reached if no accept_alternative or other delay_alternative can be selected prior to the expiration time. If several delay_alternatives have this same expiration time, one of them is selected according to the queuing policy in effect (see D.4); the default queuing policy chooses arbitrarily among the delay_alternatives whose expiration time has passed. 18
- The else part is selected and its sequence_of_statements is executed if no accept_alternative can immediately be selected; in particular, if all alternatives are closed. 19
- An open terminate_alternative is selected if the conditions stated at the end of clause 9.3 are satisfied. 20

The exception Program_Error is raised if all alternatives are closed and there is no else part. 21

NOTES

36 A selective_accept is allowed to have several open delay_alternatives. A selective_accept is allowed to have several open accept_alternatives for the same entry. 22

Examples

Example of a task body with a selective accept: 23

```
task body Server is                                                    24
   Current_Work_Item : Work_Item;
begin
   loop
      select
         accept Next_Work_Item(WI : in Work_Item) do
            Current_Work_Item := WI;
         end;
         Process_Work_Item(Current_Work_Item);
      or
         accept Shut_Down;
         exit;          -- Premature shut down requested
      or
         terminate;     -- Normal shutdown at end of scope
      end select;
   end loop;
end Server;
```

9.7.2 Timed Entry Calls

1 A timed_entry_call issues an entry call that is cancelled if the call (or a requeue-with-abort of the call) is not selected before the expiration time is reached.

Syntax

2
```
timed_entry_call ::=
 select
  entry_call_alternative
 or
  delay_alternative
 end select;
```

3
```
entry_call_alternative ::=
 entry_call_statement [sequence_of_statements]
```

Dynamic Semantics

4 For the execution of a timed_entry_call, the *entry*_name and the actual parameters are evaluated, as for a simple entry call (see 9.5.3). The expiration time (see 9.6) for the call is determined by evaluating the *delay*_expression of the delay_alternative; the entry call is then issued.

5 If the call is queued (including due to a requeue-with-abort), and not selected before the expiration time is reached, an attempt to cancel the call is made. If the call completes due to the cancellation, the optional sequence_of_statements of the delay_alternative is executed; if the entry call completes normally, the optional sequence_of_statements of the entry_call_alternative is executed.

Examples

6 *Example of a timed entry call:*

7
```
select
   Controller.Request(Medium)(Some_Item);
or
   delay 45.0;
   -- controller too busy, try something else
end select;
```

9.7.3 Conditional Entry Calls

1 A conditional_entry_call issues an entry call that is then cancelled if it is not selected immediately (or if a requeue-with-abort of the call is not selected immediately).

Syntax

2
```
conditional_entry_call ::=
 select
  entry_call_alternative
 else
  sequence_of_statements
 end select;
```

Dynamic Semantics

3 The execution of a conditional_entry_call is defined to be equivalent to the execution of a timed_entry_call with a delay_alternative specifying an immediate expiration time and the same sequence_of_statements as given after the reserved word **else**.

NOTES
37 A conditional_entry_call may briefly increase the Count attribute of the entry, even if the conditional call is not selected.

Examples

Example of a conditional entry call:

```
procedure Spin(R : in Resource) is
begin
   loop
      select
         R.Seize;
         return;
      else
         null;  -- busy waiting
      end select;
   end loop;
end;
```

9.7.4 Asynchronous Transfer of Control

An asynchronous select_statement provides asynchronous transfer of control upon completion of an entry call or the expiration of a delay.

Syntax

asynchronous_select ::=
select
triggering_alternative
then abort
abortable_part
end select;

triggering_alternative ::= triggering_statement [sequence_of_statements]

triggering_statement ::= entry_call_statement | delay_statement

abortable_part ::= sequence_of_statements

Dynamic Semantics

For the execution of an asynchronous_select whose triggering_statement is an entry_call_statement, the *entry*_name and actual parameters are evaluated as for a simple entry call (see 9.5.3), and the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started.

For the execution of an asynchronous_select whose triggering_statement is a delay_statement, the *delay*_expression is evaluated and the expiration time is determined, as for a normal delay_statement. If the expiration time has not already passed, the abortable_part is executed.

If the abortable_part completes and is left prior to completion of the triggering_statement, an attempt to cancel the triggering_statement is made. If the attempt to cancel succeeds (see 9.5.3 and 9.6), the asynchronous_select is complete.

If the triggering_statement completes other than due to cancellation, the abortable_part is aborted (if started but not yet completed — see 9.8). If the triggering_statement completes normally, the optional sequence_of_statements of the triggering_alternative is executed after the abortable_part is left.

Examples

Example of a main command loop for a command interpreter:

```
loop
    select
        Terminal.Wait_For_Interrupt;
        Put_Line("Interrupted");
    then abort
        -- This will be abandoned upon terminal interrupt
        Put_Line("-> ");
        Get_Line(Command, Last);
        Process_Command(Command(1..Last));
    end select;
end loop;
```

Example of a time-limited calculation:

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

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

An abort_statement causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. The completion of the triggering_statement of an asynchronous_select causes a sequence_of_statements to be aborted.

Syntax

abort_statement ::= **abort** *task*_name {, *task*_name};

Name Resolution Rules

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

Dynamic Semantics

For the execution of an abort_statement, the given *task*_names are evaluated in an arbitrary order. Each named task is then *aborted*, which consists of making the task *abnormal* and aborting the execution of the corresponding task_body, unless it is already completed.

When the execution of a construct is *aborted* (including that of a task_body or of a sequence_of_statements), the execution of every construct included within the aborted execution is also aborted, except for executions included within the execution of an *abort-deferred* operation; the execution of an abort-deferred operation continues to completion without being affected by the abort; the following are the abort-deferred operations:

- a protected action;
- waiting for an entry call to complete (after having initiated the attempt to cancel it — see below);
- waiting for the termination of dependent tasks;
- the execution of an Initialize procedure as the last step of the default initialization of a controlled object;
- the execution of a Finalize procedure as part of the finalization of a controlled object;

- an assignment operation to an object with a controlled part.

The last three of these are discussed further in 7.6.

When a master is aborted, all tasks that depend on that master are aborted.

The order in which tasks become abnormal as the result of an abort_statement or the abort of a sequence_of_statements is not specified by the language.

If the execution of an entry call is aborted, an immediate attempt is made to cancel the entry call (see 9.5.3). If the execution of a construct is aborted at a time when the execution is blocked, other than for an entry call, at a point that is outside the execution of an abort-deferred operation, then the execution of the construct completes immediately. For an abort due to an abort_statement, these immediate effects occur before the execution of the abort_statement completes. Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the abort_statement completes. However, the execution of the aborted construct completes no later than its next *abort completion point* (if any) that occurs outside of an abort-deferred operation; the following are abort completion points for an execution:

- the point where the execution initiates the activation of another task;
- the end of the activation of a task;
- the start or end of the execution of an entry call, accept_statement, delay_statement, or abort_statement;
- the start of the execution of a select_statement, or of the sequence_of_statements of an exception_handler.

Bounded (Run-Time) Errors

An attempt to execute an asynchronous_select as part of the execution of an abort-deferred operation is a bounded error. Similarly, an attempt to create a task that depends on a master that is included entirely within the execution of an abort-deferred operation is a bounded error. In both cases, Program_Error is raised if the error is detected by the implementation; otherwise the operations proceed as they would outside an abort-deferred operation, except that an abort of the abortable_part or the created task might or might not have an effect.

Erroneous Execution

If an assignment operation completes prematurely due to an abort, the assignment is said to be *disrupted*; the target of the assignment or its parts can become abnormal, and certain subsequent uses of the object can be erroneous, as explained in 13.9.1.

NOTES

38 An abort_statement should be used only in situations requiring unconditional termination.

39 A task is allowed to abort any task it can name, including itself.

40 Additional requirements associated with abort are given in D.6, ‘‘Preemptive Abort’’.

9.9 Task and Entry Attributes

Dynamic Semantics

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

2 T'Callable — Yields the value True when the task denoted by T is *callable*, and False otherwise; a task is callable unless it is completed or abnormal. The value of this attribute is of the predefined type Boolean.

3 T'Terminated — Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean.

4 For a prefix E that denotes an entry of a task or protected unit, the following attribute is defined. This attribute is only allowed within the body of the task or protected unit, but excluding, in the case of an entry of a task unit, within any program unit that is, itself, inner to the body of the task unit.

5 E'Count — Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*.

NOTES

6 41 For the Count attribute, the entry can be either a single entry or an entry of a family. The name of the entry or entry family can be either a direct_name or an expanded name.

7 42 Within task units, algorithms interrogating the attribute E'Count should take precautions to allow for the increase of the value of this attribute for incoming entry calls, and its decrease, for example with timed_entry_calls. Also, a conditional_entry_call may briefly increase this value, even if the conditional call is not accepted.

8 43 Within protected units, algorithms interrogating the attribute E'Count in the entry_barrier for the entry E should take precautions to allow for the evaluation of the condition of the barrier both before and after queuing a given caller.

9.10 Shared Variables

Static Semantics

1 If two different objects, including nonoverlapping parts of the same object, are *independently addressable*, they can be manipulated concurrently by two different tasks without synchronization. Normally, any two nonoverlapping objects are independently addressable. However, if packing, record layout, or Component_Size is specified for a given composite object, then it is implementation defined whether or not two nonoverlapping parts of that composite object are independently addressable.

Dynamic Semantics

2 Separate tasks normally proceed independently and concurrently with one another. However, task interactions can be used to synchronize the actions of two or more tasks to allow, for example, meaningful communication by the direct updating and reading of variables shared between the tasks. The actions of two different tasks are synchronized in this sense when an action of one task *signals* an action of the other task; an action A1 is defined to signal an action A2 under the following circumstances:

3 • If A1 and A2 are part of the execution of the same task, and the language rules require A1 to be performed before A2;

4 • If A1 is the action of an activator that initiates the activation of a task, and A2 is part of the execution of the task that is activated;

5 • If A1 is part of the activation of a task, and A2 is the action of waiting for completion of the activation;

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

7 • If A1 is the action of issuing an entry call, and A2 is part of the corresponding execution of the appropriate entry_body or accept_statement.

8 • If A1 is part of the execution of an accept_statement or entry_body, and A2 is the action of returning from the corresponding entry call;

- If A1 is part of the execution of a protected procedure body or entry_body for a given protected object, and A2 is part of a later execution of an entry_body for the same protected object;
- If A1 signals some action that in turn signals A2.

Erroneous Execution

Given an action of assigning to an object, and an action of reading or updating a part of the same object (or of a neighboring object if the two are not independently addressable), then the execution of the actions is erroneous unless the actions are *sequential*. Two actions are sequential if one of the following is true:

- One action signals the other;
- Both actions occur as part of the execution of the same task;
- Both actions occur as part of protected actions on the same protected object, and at most one of the actions is part of a call on a protected function of the protected object.

A pragma Atomic or Atomic_Components may also be used to ensure that certain reads and updates are sequential — see C.6.

9.11 Example of Tasking and Synchronization

Examples

The following example defines a buffer protected object to smooth variations between the speed of output of a producing task and the speed of input of some consuming task. For instance, the producing task might have the following structure:

```
task Producer;
task body Producer is
   Char : Character;
begin
   loop
      ... -- produce the next character Char
      Buffer.Write(Char);
      exit when Char = ASCII.EOT;
   end loop;
end Producer;
```

and the consuming task might have the following structure:

```
task Consumer;
task body Consumer is
   Char : Character;
begin
   loop
      Buffer.Read(Char);
      exit when Char = ASCII.EOT;
      ... -- consume the character Char
   end loop;
end Consumer;
```

The buffer object contains an internal pool of characters managed in a round-robin fashion. The pool has two indices, an In_Index denoting the space for the next input character and an Out_Index denoting the space for the next output character.

8
```
protected Buffer is
   entry Read (C : out Character);
   entry Write(C : in  Character);
private
   Pool      : String(1 .. 100);
   Count     : Natural := 0;
   In_Index, Out_Index : Positive := 1;
end Buffer;
```

9
```
protected body Buffer is
   entry Write(C : in Character)
      when Count < Pool'Length is
   begin
      Pool(In_Index) := C;
      In_Index := (In_Index mod Pool'Length) + 1;
      Count    := Count + 1;
   end Write;
```

10
```
   entry Read(C : out Character)
      when Count > 0 is
   begin
      C := Pool(Out_Index);
      Out_Index := (Out_Index mod Pool'Length) + 1;
      Count     := Count - 1;
   end Read;
end Buffer;
```

Section 13: Representation Issues

This section describes features for querying and controlling aspects of representation and for interfacing to hardware. 1

13.1 Representation Items

There are three kinds of *representation items*: representation_clauses, component_clauses, and *representation pragmas*. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware). Representation items also specify other specifiable properties of entities. A representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation. 1

Syntax

representation_clause ::= attribute_definition_clause 2
| enumeration_representation_clause
| record_representation_clause
| at_clause

local_name ::= direct_name 3
| direct_name'attribute_designator
| *library_unit*_name

A representation pragma is allowed only at places where a representation_clause or compilation_unit is allowed. 4

Name Resolution Rules

In a representation item, if the local_name is a direct_name, then it shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative_region as the representation item. If the local_name has an attribute_designator, then it shall resolve to denote an implementation-defined component (see 13.5.1) or a class-wide type implicitly declared immediately within the same declarative_region as the representation item. A local_name that is a *library_unit*_name (only permitted in a representation pragma) shall resolve to denote the library_item that immediately precedes (except for other pragmas) the representation pragma. 5

Legality Rules

The local_name of a representation_clause or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same declarative_part, package_specification, task_definition, protected_definition, or record_definition as the representation item. If a local_name denotes a local callable entity, it may do so through a local subprogram_renaming_declaration (as a way to resolve ambiguity in the presence of overloading); otherwise, the local_name shall not denote a renaming_declaration. 6

The *representation* of an object consists of a certain number of bits (the *size* of the object). These are the bits that are normally read or updated by the machine code when loading, storing, or operating-on the value of the object. This includes some padding bits, when the size of the object is greater than the size of its subtype. Such padding bits are considered to be part of the representation of the object, rather than being gaps between objects, if these bits are normally read and updated. 7

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

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

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

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

12 A representation item that specifies the Size for a given subtype, or the size or storage place for an object (including a component) of a given subtype, shall allow for enough storage space to accommodate any value of the subtype.

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

Static Semantics

14 If two subtypes statically match, then their subtype-specific aspects (Size and Alignment) are the same.

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

16 Each aspect of representation of an entity is as follows:

17 • If the aspect is *specified* for the entity, meaning that it is either directly specified or inherited, then that aspect of the entity is as specified, except in the case of Storage_Size, which specifies a minimum.

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

Dynamic Semantics

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

Implementation Permissions

20 An implementation may interpret aspects of representation in an implementation-defined manner. An implementation may place implementation-defined restrictions on representation items. A *recommended level of support* is specified for representation items and related features in each subclause. These recommendations are changed to requirements for implementations that support the Systems Programming Annex (see C.2, ''Required Representation Support'').

Implementation Advice

The recommended level of support for all representation items is qualified as follows: 21

- An implementation need not support representation items containing nonstatic expressions, except that an implementation should support a representation item for a given entity if each nonstatic expression in the representation item is a name that statically denotes a constant declared before the entity. 22
- An implementation need not support a specification for the Size for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints. 23
- An aliased component, or a component whose type is by-reference, should always be allocated at an addressable location. 24

13.2 Pragma Pack

A pragma Pack specifies that storage minimization should be the main criterion when selecting the representation of a composite type. 1

Syntax

The form of a pragma Pack is as follows: 2

pragma Pack(*first_subtype*_local_name); 3

Legality Rules

The *first_subtype*_local_name of a pragma Pack shall denote a composite subtype. 4

Static Semantics

A pragma Pack specifies the *packing* aspect of representation; the type (or the extension part) is said to be *packed*. For a type extension, the parent part is packed as for the parent type, and a pragma Pack causes packing only of the extension part. 5

Implementation Advice

If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations. 6

The recommended level of support for pragma Pack is: 7

- For a packed record type, the components should be packed as tightly as possible subject to the Sizes of the component subtypes, and subject to any record_representation_clause that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words. 8
- For a packed array type, if the component subtype's Size is less than or equal to the word size, and Component_Size is not specified for the type, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size. 9

13.3 Representation Attributes

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

Syntax

2
```
attribute_definition_clause ::=
    for local_name'attribute_designator use expression;
  | for local_name'attribute_designator use name;
```

Name Resolution Rules

3 For an attribute_definition_clause that specifies an attribute that denotes a value, the form with an expression shall be used. Otherwise, the form with a name shall be used.

4 For an attribute_definition_clause that specifies an attribute that denotes a value or an object, the expected type for the expression or name is that of the attribute. For an attribute_definition_clause that specifies an attribute that denotes a subprogram, the expected profile for the name is the profile required for the attribute. For an attribute_definition_clause that specifies an attribute that denotes some other kind of entity, the name shall resolve to denote an entity of the appropriate kind.

Legality Rules

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

6 For an attribute_definition_clause that specifies an attribute that denotes a subprogram, the profile shall be mode conformant with the one required for the attribute, and the convention shall be Ada. Additional requirements are defined for particular attributes.

Static Semantics

7 A *Size clause* is an attribute_definition_clause whose attribute_designator is Size. Similar definitions apply to the other specifiable attributes.

8 A *storage element* is an addressable element of storage in the machine. A *word* is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of an integral number of storage elements.

9 The following attributes are defined:

10 For a prefix X that denotes an object, program unit, or label:

11 X'Address — Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address.

12 Address may be specified for stand-alone objects and for program units via an attribute_definition_clause.

Erroneous Execution

13 If an Address is specified, it is the programmer's responsibility to ensure that the address is valid; otherwise, program execution is erroneous.

Implementation Advice

For an array X, X'Address should point at the first component of the array, and not at the array bounds.

The recommended level of support for the Address attribute is:

- X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified.
- An implementation should support Address clauses for imported subprograms.
- Objects (including subcomponents) that are aliased or of a by-reference type should be allocated on storage element boundaries.
- If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases.

NOTES

1 The specification of a link name in a pragma Export (see B.1) for a subprogram or object is an alternative to explicit specification of its link-time address, allowing a link-time directive to place the subprogram or object within memory.

2 The rules for the Size attribute imply, for an aliased object X, that if X'Size = Storage_Unit, then X'Address points at a storage element containing all of the bits of X, and only the bits of X.

Static Semantics

For a prefix X that denotes a subtype or object:

X'Alignment — The Address of an object that is allocated under control of the implementation is an integral multiple of the Alignment of the object (that is, the Address modulo the Alignment is zero). The offset of a record component is a multiple of the Alignment of the component. For an object that is not allocated under control of the implementation (that is, one that is imported, that is allocated by a user-defined allocator, whose Address has been specified, or is designated by an access value returned by an instance of Unchecked_Conversion), the implementation may assume that the Address is an integral multiple of its Alignment. The implementation shall not assume a stricter alignment.

The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary.

Alignment may be specified for first subtypes and stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. If the Alignment of a subtype is specified, then the Alignment of an object of the subtype is at least as strict, unless the object's Alignment is also specified. The Alignment of an object created by an allocator is that of the designated subtype.

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.

Erroneous Execution

Program execution is erroneous if an Address clause is given that conflicts with the Alignment.

If the Alignment is specified for an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to the Alignment.

Implementation Advice

The recommended level of support for the Alignment attribute for subtypes is:

- An implementation should support specified Alignments that are factors and multiples of the number of storage elements per word, subject to the following:

- An implementation need not support specified Alignments for combinations of Sizes and Alignments that cannot be easily loaded and stored by available machine instructions.
- An implementation need not support specified Alignments that are greater than the maximum Alignment the implementation ever returns by default.

The recommended level of support for the Alignment attribute for objects is:

- Same as above, for subtypes, but in addition:
- For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

NOTES

3 Alignment is a subtype-specific attribute.

4 The Alignment of a composite object is always equal to the least common multiple of the Alignments of its components, or a multiple thereof.

5 A component_clause, Component_Size clause, or a pragma Pack can override a specified Alignment.

Static Semantics

For a prefix X that denotes an object:

X'Size — Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*.

Size may be specified for stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative.

Implementation Advice

The recommended level of support for the Size attribute of objects is:

- A Size clause should be supported for an object if the specified Size is at least as large as its subtype's Size, and corresponds to a size in storage elements that is a multiple of the object's Alignment (if the Alignment is nonzero).

Static Semantics

For every subtype S:

S'Size — If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S:

- A record component of subtype S when the record type is packed.
- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype.

If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. The Size of an object is at least as large as that of its subtype, unless the object's Size is determined by a Size clause, a component_clause, or a Component_Size clause. Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative.

Implementation Requirements

In an implementation, Boolean'Size shall be 1.

Implementation Advice

If the Size of a subtype is specified, and allows for efficient independent addressability (see 9.10) on the target architecture, then the Size of the following objects of the subtype should equal the Size of the subtype:

- Aliased objects (including components).
- Unaliased components, unless the Size of the component is determined by a component_clause or Component_Size clause.

A Size clause on a composite subtype should not affect the internal layout of components.

The recommended level of support for the Size attribute of subtypes is:

- The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits needed to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation.
- For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.

NOTES

6 Size is a subtype-specific attribute.

7 A component_clause or Component_Size clause can override a specified Size. A pragma Pack cannot.

Static Semantics

For a prefix T that denotes a task object (after any implicit dereference):

T'Storage_Size — Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the ''task control block'' used by some implementations.) If a pragma Storage_Size is given, the value of the Storage_Size attribute is at least the value specified in the pragma.

A pragma Storage_Size specifies the amount of storage to be reserved for the execution of a task.

Syntax

The form of a pragma Storage_Size is as follows:

pragma Storage_Size(expression);

A pragma Storage_Size is allowed only immediately within a task_definition.

Name Resolution Rules

The expression of a pragma Storage_Size is expected to be of any integer type.

Dynamic Semantics

A pragma Storage_Size is elaborated when an object of the type defined by the immediately enclosing task_definition is created. For the elaboration of a pragma Storage_Size, the expression is evaluated; the Storage_Size attribute of the newly created task object is at least the value of the expression.

At the point of task object creation, or upon task activation, Storage_Error is raised if there is insufficient free storage to accommodate the requested Storage_Size.

Static Semantics

68 For a prefix X that denotes an array subtype or array object (after any implicit dereference):

69 X'Component_Size

Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*.

70 Component_Size may be specified for array types via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative.

Implementation Advice

71 The recommended level of support for the Component_Size attribute is:

72 • An implementation need not support specified Component_Sizes that are less than the Size of the component subtype.

73 • An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps between components. For other Component_Sizes (if supported), the array should contain no gaps between components when packing is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation.

Static Semantics

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

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

Implementation Requirements

76 In an implementation, the default external tag for each specific tagged type declared in a partition shall be distinct, so long as the type is declared outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined. At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions.

NOTES

77 8 The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Size, Component_Size, Alignment, External_Tag, Small, Bit_Order, Storage_Pool, Storage_Size, Write, Output, Read, Input, and Machine_Radix.

78 9 It follows from the general rules in 13.1 that if one writes ''**for** X'Size **use** Y;'' then the X'Size attribute_reference will return Y (assuming the implementation allows the Size clause). The same is true for all of the specifiable attributes except Storage_Size.

Examples

79 *Examples of attribute definition clauses:*

80
```
Byte : constant := 8;
Page : constant := 2**12;
```

81
```
type Medium is range 0 .. 65_000;
for Medium'Size use 2*Byte;
for Medium'Alignment use 2;
Device_Register : Medium;
for Device_Register'Size use Medium'Size;
for Device_Register'Address use System.Storage_Elements.To_Address(16#FFFF_0020#);
```

82
```
type Short is delta 0.01 range -100.0 .. 100.0;
for Short'Size use 15;
```

```
for Car_Name'Storage_Size use -- specify access type's storage pool size
        2000*((Car'Size/System.Storage_Unit) +1); -- approximately 2000 cars
function My_Read(Stream : access Ada.Streams.Root_Stream_Type'Class)
  return T;
for T'Read use My_Read; -- see 13.13.2
```

NOTES

10 *Notes on the examples:* In the Size clause for Short, fifteen bits is the minimum necessary, since the type definition requires Short'Small <= 2**(–7).

13.4 Enumeration Representation Clauses

An enumeration_representation_clause specifies the internal codes for enumeration literals.

Syntax

enumeration_representation_clause ::=
  for *first_subtype*_local_name **use** enumeration_aggregate;

enumeration_aggregate ::= array_aggregate

Name Resolution Rules

The enumeration_aggregate shall be written as a one-dimensional array_aggregate, for which the index subtype is the unconstrained subtype of the enumeration type, and each component expression is expected to be of any integer type.

Legality Rules

The *first_subtype*_local_name of an enumeration_representation_clause shall denote an enumeration subtype.

The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type.

Static Semantics

An enumeration_representation_clause specifies the *coding* aspect of representation. The coding consists of the *internal code* for each enumeration literal, that is, the integral value used internally to represent each literal.

Implementation Requirements

For nonboolean enumeration types, if the coding is not specified for the type, then for each value of the type, the internal code shall be equal to its position number.

Implementation Advice

The recommended level of support for enumeration_representation_clauses is:

- An implementation should support at least the internal codes in the range System.Min_Int..System.Max_Int. An implementation need not support enumeration_representation_clauses for boolean types.

NOTES

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

Examples

12 *Example of an enumeration representation clause:*

```
13     type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
14     for Mix_Code use
          (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ =>33);
```

13.5 Record Layout

1 The *(record) layout* aspect of representation consists of the *storage places* for some or all components, that is, storage place attributes of the components. The layout can be specified with a record_representation_clause.

13.5.1 Record Representation Clauses

1 A record_representation_clause specifies the storage representation of records and record extensions, that is, the order, position, and size of components (including discriminants, if any).

Syntax

2 record_representation_clause ::=
for *first_subtype*_local_name **use**
record [mod_clause]
{component_clause}
end record;

3 component_clause ::=
*component*_local_name **at** position **range** first_bit .. last_bit;

4 position ::= *static*_expression

5 first_bit ::= *static*_simple_expression

6 last_bit ::= *static*_simple_expression

Name Resolution Rules

7 Each position, first_bit, and last_bit is expected to be of any integer type.

Legality Rules

8 The *first_subtype*_local_name of a record_representation_clause shall denote a specific nonlimited record or record extension subtype.

9 If the *component*_local_name is a direct_name, the local_name shall denote a component of the type. For a record extension, the component shall not be inherited, and shall not be a discriminant that corresponds to a discriminant of the parent type. If the *component*_local_name has an attribute_designator, the direct_name of the local_name shall denote either the declaration of the type or a component of the type, and the attribute_designator shall denote an implementation-defined implicit component of the type.

10 The position, first_bit, and last_bit shall be static expressions. The value of position and first_bit shall be nonnegative. The value of last_bit shall be no less than first_bit – 1.

11 At most one component_clause is allowed for each component of the type, including for each discriminant (component_clauses may be given for some, all, or none of the components). Storage places within a component_list shall not overlap, unless they are for components in distinct variants of the same variant_part.

A name that denotes a component of a type is not allowed within a record_representation_clause for the type, except as the *component*_local_name of a component_clause. 12

Static Semantics

A record_representation_clause (without the mod_clause) specifies the layout. The storage place attributes (see 13.5.2) are taken from the values of the position, first_bit, and last_bit expressions after normalizing those values so that first_bit is less than Storage_Unit. 13

A record_representation_clause for a record extension does not override the layout of the parent part; if the layout was specified for the parent type, it is inherited by the record extension. 14

Implementation Permissions

An implementation may generate implementation-defined components (for example, one containing the offset of another component). An implementation may generate names that denote such implementation-defined components; such names shall be implementation-defined attribute_references. An implementation may allow such implementation-defined names to be used in record_representation_clauses. An implementation can restrict such component_clauses in any manner it sees fit. 15

If a record_representation_clause is given for an untagged derived type, the storage place attributes for all of the components of the derived type may differ from those of the corresponding components of the parent type, even for components whose storage place is not specified explicitly in the record_representation_clause. 16

Implementation Advice

The recommended level of support for record_representation_clauses is: 17

- An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model. 18
- A storage place should be supported if its size is equal to the Size of the component subtype, and it starts and ends on a boundary that obeys the Alignment of the component subtype. 19
- If the default bit ordering applies to the declaration of a given type, then for a component whose subtype's Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported. 20
- An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place. 21
- An implementation need not support a component_clause for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified. 22

NOTES

12 If no component_clause is given for a component, then the choice of the storage place for the component is left to the implementation. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation. 23

Examples

Example of specifying the layout of a record type: 24

```
Word : constant := 4;  -- storage element is byte, 4 bytes per word          25

type State         is (A,M,W,P);                                             26
type Mode          is (Fix, Dec, Exp, Signif);

type Byte_Mask     is array (0..7)  of Boolean;                              27
type State_Mask    is array (State) of Boolean;
type Mode_Mask     is array (Mode)  of Boolean;
```

```
type Program_Status_Word is
  record
      System_Mask        : Byte_Mask;
      Protection_Key     : Integer range 0 .. 3;
      Machine_State      : State_Mask;
      Interrupt_Cause    : Interruption_Code;
      Ilc                : Integer range 0 .. 3;
      Cc                 : Integer range 0 .. 3;
      Program_Mask       : Mode_Mask;
      Inst_Address       : Address;
end record;
for Program_Status_Word use
  record
      System_Mask      at 0*Word range 0  .. 7;
      Protection_Key   at 0*Word range 10 .. 11; -- bits 8,9 unused
      Machine_State    at 0*Word range 12 .. 15;
      Interrupt_Cause  at 0*Word range 16 .. 31;
      Ilc              at 1*Word range 0  .. 1;  -- second word
      Cc               at 1*Word range 2  .. 3;
      Program_Mask     at 1*Word range 4  .. 7;
      Inst_Address     at 1*Word range 8  .. 31;
  end record;
for Program_Status_Word'Size use 8*System.Storage_Unit;
for Program_Status_Word'Alignment use 8;
```

NOTES

13 *Note on the example:* The record_representation_clause defines the record layout. The Size clause guarantees that (at least) eight storage elements are used for objects of the type. The Alignment clause guarantees that aliased, imported, or exported objects of the type will have addresses divisible by eight.

13.5.2 Storage Place Attributes

Static Semantics

For a component C of a composite, non-array object R, the *storage place attributes* are defined:

R.C'Position — Denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*.

R.C'First_Bit — Denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*.

R.C'Last_Bit — Denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*.

Implementation Advice

If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes.

13.5.3 Bit Ordering

The Bit_Order attribute specifies the interpretation of the storage place attributes.

Static Semantics

A bit ordering is a method of interpreting the meaning of the storage place attributes. High_Order_First (known in the vernacular as ''big endian'') means that the first bit of a storage element (bit 0) is the most significant bit (interpreting the sequence of bits that represent a component as an unsigned integer value).

Low_Order_First (known in the vernacular as ‘‘little endian’’) means the opposite: the first bit is the least significant.

For every specific record subtype S, the following attribute is defined:

S’Bit_Order — Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. Bit_Order may be specified for specific record types via an attribute_definition_clause; the expression of such a clause shall be static.

If Word_Size = Storage_Unit, the default bit ordering is implementation defined. If Word_Size > Storage_Unit, the default bit ordering is the same as the ordering of storage elements in a word, when interpreted as an integer.

The storage place attributes of a component of a type are interpreted according to the bit ordering of the type.

Implementation Advice

The recommended level of support for the nondefault bit ordering is:

- If Word_Size = Storage_Unit, then the implementation should support the nondefault bit ordering in addition to the default bit ordering.

13.6 Change of Representation

A type_conversion (see 4.6) can be used to convert between two different representations of the same array or record. To convert an array from one representation to another, two array types need to be declared with matching component subtypes, and convertible index types. If one type has packing specified and the other does not, then explicit conversion can be used to pack or unpack an array.

To convert a record from one representation to another, two record types with a common ancestor type need to be declared, with no inherited subprograms. Distinct representations can then be specified for the record types, and explicit conversion between the types can be used to effect a change in representation.

Examples

Example of change of representation:

```
-- Packed_Descriptor and Descriptor are two different types
-- with identical characteristics, apart from their
-- representation
type Descriptor is
    record
      -- components of a descriptor
    end record;
type Packed_Descriptor is new Descriptor;
for Packed_Descriptor use
    record
      -- component clauses for some or for all components
    end record;
-- Change of representation can now be accomplished by explicit type conversions:
D : Descriptor;
P : Packed_Descriptor;
P := Packed_Descriptor(D);  -- pack D
D := Descriptor(P);         -- unpack P
```

13.7 The Package System

1 For each implementation there is a library package called System which includes the definitions of certain configuration-dependent characteristics.

Static Semantics

2 The following language-defined library package exists:

3
```
package System is
   pragma Preelaborate(System);
```

4
```
   type Name is implementation-defined-enumeration-type;
   System_Name : constant Name := implementation-defined;
```

5
```
   -- System-Dependent Named Numbers:
```

6
```
   Min_Int               : constant := root_integer'First;
   Max_Int               : constant := root_integer'Last;
```

7
```
   Max_Binary_Modulus    : constant := implementation-defined;
   Max_Nonbinary_Modulus : constant := implementation-defined;
```

8
```
   Max_Base_Digits       : constant := root_real'Digits;
   Max_Digits            : constant := implementation-defined;
```

9
```
   Max_Mantissa          : constant := implementation-defined;
   Fine_Delta            : constant := implementation-defined;
```

10
```
   Tick                  : constant := implementation-defined;
```

11
```
   -- Storage-related Declarations:
```

12
```
   type Address is implementation-defined;
   Null_Address : constant Address;
```

13
```
   Storage_Unit : constant := implementation-defined;
   Word_Size    : constant := implementation-defined * Storage_Unit;
   Memory_Size  : constant := implementation-defined;
```

14
```
   -- Address Comparison:
   function "<" (Left, Right : Address) return Boolean;
   function "<="(Left, Right : Address) return Boolean;
   function ">" (Left, Right : Address) return Boolean;
   function ">="(Left, Right : Address) return Boolean;
   function "=" (Left, Right : Address) return Boolean;
-- function "/=" (Left, Right : Address) return Boolean;
   -- "/=" is implicitly defined
   pragma Convention(Intrinsic, "<");
   ... -- and so on for all language-defined subprograms in this package
```

15
```
   -- Other System-Dependent Declarations:
   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order;
```

16
```
   -- Priority-related declarations (see D.1):
   subtype Any_Priority is Integer range implementation-defined;
   subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;
   subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last;
```

17
```
   Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;
```

18
```
private
   ... -- not specified by the language
end System;
```

19 Name is an enumeration subtype. Values of type Name are the names of alternative machine configurations handled by the implementation. System_Name represents the current machine configuration.

The named numbers Fine_Delta and Tick are of the type *universal_real*; the others are of the type *universal_integer*. 20

The meanings of the named numbers are: 21

Min_Int — The smallest (most negative) value allowed for the expressions of a signed_integer_type_definition. 22

Max_Int — The largest (most positive) value allowed for the expressions of a signed_integer_type_definition. 23

Max_Binary_Modulus — A power of two such that it, and all lesser positive powers of two, are allowed as the modulus of a modular_type_definition. 24

Max_Nonbinary_Modulus — A value such that it, and all lesser positive integers, are allowed as the modulus of a modular_type_definition. 25

Max_Base_Digits — The largest value allowed for the requested decimal precision in a floating_point_definition. 26

Max_Digits — The largest value allowed for the requested decimal precision in a floating_point_definition that has no real_range_specification. Max_Digits is less than or equal to Max_Base_Digits. 27

Max_Mantissa — The largest possible number of binary digits in the mantissa of machine numbers of a user-defined ordinary fixed point type. (The mantissa is defined in Annex G.) 28

Fine_Delta — The smallest delta allowed in an ordinary_fixed_point_definition that has the real_range_specification **range** –1.0 .. 1.0. 29

Tick — A period in seconds approximating the real time interval during which the value of Calendar.Clock remains constant. 30

Storage_Unit — The number of bits per storage element. 31

Word_Size — The number of bits per word. 32

Memory_Size — An implementation-defined value that is intended to reflect the memory size of the configuration in storage elements. 33

Address is of a definite, nonlimited type. Address represents machine addresses capable of addressing individual storage elements. Null_Address is an address that is distinct from the address of any object or program unit. 34

See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order. 35

Implementation Permissions

An implementation may add additional implementation-defined declarations to package System and its children. However, it is usually better for the implementation to provide additional functionality via implementation-defined children of System. Package System may be declared pure. 36

Implementation Advice

Address should be of a private type. 37

NOTES

14 There are also some language-defined child packages of System defined elsewhere. 38

13.7.1 The Package System.Storage_Elements

Static Semantics

1 The following language-defined library package exists:

2
```
package System.Storage_Elements is
   pragma Preelaborate(System.Storage_Elements);
```
3
```
   type Storage_Offset is range implementation-defined;
```
4
```
   subtype Storage_Count is Storage_Offset range 0..Storage_Offset'Last;
```
5
```
   type Storage_Element is mod implementation-defined;
   for Storage_Element'Size use Storage_Unit;
   type Storage_Array is array
     (Storage_Offset range <>) of aliased Storage_Element;
   for Storage_Array'Component_Size use Storage_Unit;
```
6
```
   -- Address Arithmetic:
```
7
```
   function "+"(Left : Address; Right : Storage_Offset)
     return Address;
   function "+"(Left : Storage_Offset; Right : Address)
     return Address;
   function "-"(Left : Address; Right : Storage_Offset)
     return Address;
   function "-"(Left, Right : Address)
     return Storage_Offset;
```
8
```
   function "mod"(Left : Address; Right : Storage_Offset)
     return Storage_Offset;
```
9
```
   -- Conversion to/from integers:
```
10
```
   type Integer_Address is implementation-defined;
   function To_Address(Value : Integer_Address) return Address;
   function To_Integer(Value : Address) return Integer_Address;
```
11
```
   pragma Convention(Intrinsic, "+");
      -- ...and so on for all language-defined subprograms declared in this package.
end System.Storage_Elements;
```

12 Storage_Element represents a storage element. Storage_Offset represents an offset in storage elements. Storage_Count represents a number of storage elements. Storage_Array represents a contiguous sequence of storage elements.

13 Integer_Address is a (signed or modular) integer subtype. To_Address and To_Integer convert back and forth between this type and Address.

Implementation Requirements

14 Storage_Offset'Last shall be greater than or equal to Integer'Last or the largest possible storage offset, whichever is smaller. Storage_Offset'First shall be <= (–Storage_Offset'Last).

Implementation Permissions

15 Package System.Storage_Elements may be declared pure.

Implementation Advice

16 Operations in System and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to ''wrap around.'' Operations that do not make sense should raise Program_Error.

13.7.2 The Package System.Address_To_Access_Conversions

Static Semantics

1 The following language-defined generic library package exists:

```
generic                                                                    2
    type Object(<>) is limited private;
package System.Address_To_Access_Conversions is
   pragma Preelaborate(Address_To_Access_Conversions);
   type Object_Pointer is access all Object;                               3
   function To_Pointer(Value : Address) return Object_Pointer;
   function To_Address(Value : Object_Pointer) return Address;
   pragma Convention(Intrinsic, To_Pointer);                               4
   pragma Convention(Intrinsic, To_Address);
end System.Address_To_Access_Conversions;
```

The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address. To_Pointer(X'Address) is equal to X'Unchecked_Access for any X that allows Unchecked_Access. To_Pointer(Null_Address) returns **null**. For other addresses, the behavior is unspecified. To_Address(**null**) returns Null_Address (for **null** of the appropriate type). To_Address(Y), where Y /= **null**, returns Y.**all**'Address. 5

Implementation Permissions

An implementation may place restrictions on instantiations of Address_To_Access_Conversions. 6

13.8 Machine Code Insertions

A machine code insertion can be achieved by a call to a subprogram whose sequence_of_statements contains code_statements. 1

Syntax

code_statement ::= qualified_expression; 2

A code_statement is only allowed in the handled_sequence_of_statements of a subprogram_body. If a subprogram_body contains any code_statements, then within this subprogram_body the only allowed form of statement is a code_statement (labeled or not), the only allowed declarative_items are use_clauses, and no exception_handler is allowed (comments and pragmas are allowed as usual). 3

Name Resolution Rules

The qualified_expression is expected to be of any type. 4

Legality Rules

The qualified_expression shall be of a type declared in package System.Machine_Code. 5

A code_statement shall appear only within the scope of a with_clause that mentions package System.Machine_Code. 6

Static Semantics

The contents of the library package System.Machine_Code (if provided) are implementation defined. The meaning of code_statements is implementation defined. Typically, each qualified_expression represents a machine instruction or assembly directive. 7

Implementation Permissions

An implementation may place restrictions on code_statements. An implementation is not required to provide package System.Machine_Code. 8

NOTES

9 15 An implementation may provide implementation-defined pragmas specifying register conventions and calling conventions.

10 16 Machine code functions are exempt from the rule that a return_statement is required. In fact, return_statements are forbidden, since only code_statements are allowed.

11 17 Intrinsic subprograms (see 6.3.1, ''Conformance Rules'') can also be used to achieve machine code insertions. Interface to assembly language can be achieved using the features in Annex B, ''Interface to Other Languages''.

Examples

12 *Example of a code statement:*

13
```
M : Mask;
procedure Set_Mask; pragma Inline(Set_Mask);
```

14
```
procedure Set_Mask is
  use System.Machine_Code; -- assume ''with System.Machine_Code;'' appears somewhere above
begin
  SI_Format'(Code => SSM, B => M'Base_Reg, D => M'Disp);
  -- Base_Reg and Disp are implementation-defined attributes
end Set_Mask;
```

13.9 Unchecked Type Conversions

1 An unchecked type conversion can be achieved by a call to an instance of the generic function Unchecked_Conversion.

Static Semantics

2 The following language-defined generic library function exists:

3
```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);
```

Dynamic Semantics

4 The size of the formal parameter S in an instance of Unchecked_Conversion is that of its subtype. This is the actual subtype passed to Source, except when the actual is an unconstrained composite subtype, in which case the subtype is constrained by the bounds or discriminants of the value of the actual expression passed to S.

5 If all of the following are true, the effect of an unchecked conversion is to return the value of an object of the target subtype whose representation is the same as that of the source object S:

6 • S'Size = Target'Size.

7 • S'Alignment = Target'Alignment.

8 • The target subtype is not an unconstrained composite subtype.

9 • S and the target subtype both have a contiguous representation.

10 • The representation of S is a representation of an object of the target subtype.

11 Otherwise, the effect is implementation defined; in particular, the result can be abnormal (see 13.9.1).

Implementation Permissions

12 An implementation may return the result of an unchecked conversion by reference, if the Source type is not a by-copy type. In this case, the result of the unchecked conversion represents simply a different (read-only) view of the operand of the conversion.

An implementation may place restrictions on Unchecked_Conversion. 13

Implementation Advice

The Size of an array object should not include its bounds; hence, the bounds should not be part of the converted data. 14

The implementation should not generate unnecessary run-time checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment. 15

The recommended level of support for unchecked conversions is: 16

- Unchecked conversions should be supported and should be reversible in the cases where this clause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph. 17

13.9.1 Data Validity

Certain actions that can potentially lead to erroneous execution are not directly erroneous, but instead can cause objects to become *abnormal*. Subsequent uses of abnormal objects can be erroneous. 1

A scalar object can have an *invalid representation*, which means that the object's representation does not represent any value of the object's subtype. The primary cause of invalid representations is uninitialized variables. 2

Abnormal objects and invalid representations are explained in this subclause. 3

Dynamic Semantics

When an object is first created, and any explicit or default initializations have been performed, the object and all of its parts are in the *normal* state. Subsequent operations generally leave them normal. However, an object or part of an object can become *abnormal* in the following ways: 4

- An assignment to the object is disrupted due to an abort (see 9.8) or due to the failure of a language-defined check (see 11.6). 5
- The object is not scalar, and is passed to an **in out** or **out** parameter of an imported procedure or language-defined input procedure, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype. 6

Whether or not an object actually becomes abnormal in these cases is not specified. An abnormal object becomes normal again upon successful completion of an assignment to the object as a whole. 7

Erroneous Execution

It is erroneous to evaluate a primary that is a name denoting an abnormal object, or to evaluate a prefix that denotes an abnormal object. 8

Bounded (Run-Time) Errors

If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an *invalid representation*. It is a bounded error to evaluate the value of such an object. If the error is detected, either Constraint_Error or Program_Error is 9

raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations. The semantics of operations on invalid representations are as follows:

- If the representation of the object represents a value of the object's type, the value of the type is used.
- If the representation of the object does not represent a value of the object's type, the semantics of operations on such representations is implementation-defined, but does not by itself lead to erroneous or unpredictable execution, or to other objects becoming abnormal.

Erroneous Execution

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.

The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an access-to-variable value that designates a constant object. Such an access value can exist, for example, because of Unchecked_Deallocation, Unchecked_Access, or Unchecked_Conversion.

NOTES

18 Objects can become abnormal due to other kinds of actions that directly update the object's representation; such actions are generally considered directly erroneous, however.

13.9.2 The Valid Attribute

The Valid attribute can be used to check the validity of data produced by unchecked conversion, input, interface to foreign languages, and the like.

Static Semantics

For a prefix X that denotes a scalar object (after any implicit dereference), the following attribute is defined:

X'Valid — Yields True if and only if the object denoted by X is normal and has a valid representation. The value of this attribute is of the predefined type Boolean.

NOTES

19 Invalid data can be created in the following cases (not counting erroneous or unpredictable execution):

- an uninitialized scalar object,
- the result of an unchecked conversion,
- input,
- interface to another language (including machine code),
- aborting an assignment,
- disrupting an assignment due to the failure of a language-defined check (see 11.6), and
- use of an object whose Address has been specified.

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

13.10 Unchecked Access Value Creation

The attribute Unchecked_Access is used to create access values in an unsafe manner — the programmer is responsible for preventing ''dangling references.''

Annex C
(normative)

Systems Programming

The Systems Programming Annex specifies additional capabilities provided for low-level programming. These capabilities are also required in many real-time, embedded, distributed, and information systems. 1

C.1 Access to Machine Operations

This clause specifies rules regarding access to machine instructions from within an Ada program. 1

Implementation Requirements

The implementation shall support machine code insertions (see 13.8) or intrinsic subprograms (see 6.3.1) (or both). Implementation-defined attributes shall be provided to allow the use of Ada entities as operands. 2

Implementation Advice

The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any. 3

The interfacing pragmas (see Annex B) should support interface to assembler; the default assembler should be associated with the convention identifier Assembler. 4

If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported. 5

Documentation Requirements

The implementation shall document the overhead associated with calling machine-code or intrinsic subprograms, as compared to a fully-inlined call, and to a regular out-of-line call. 6

The implementation shall document the types of the package System.Machine_Code usable for machine code insertions, and the attributes to be used in machine code insertions for references to Ada entities. 7

The implementation shall document the subprogram calling conventions associated with the convention identifiers available for use with the interfacing pragmas (Ada and Assembler, at a minimum), including register saving, exception propagation, parameter passing, and function value returning. 8

For exported and imported subprograms, the implementation shall document the mapping between the Link_Name string, if specified, or the Ada designator, if not, and the external link name used for such a subprogram. 9

Implementation Advice

The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms. 10

11 It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include:

12 • Atomic read-modify-write operations — e.g., test and set, compare and swap, decrement and test, enqueue/dequeue.

13 • Standard numeric functions — e.g., *sin*, *log*.

14 • String manipulation operations — e.g., translate and test.

15 • Vector operations — e.g., compare vector against thresholds.

16 • Direct operations on I/O ports.

C.2 Required Representation Support

1 This clause specifies minimal requirements on the implementation's support for representation items and related features.

Implementation Requirements

2 The implementation shall support at least the functionality defined by the recommended levels of support in Section 13.

C.3 Interrupt Support

1 This clause specifies the language-defined model for hardware interrupts in addition to mechanisms for handling interrupts.

Dynamic Semantics

2 An *interrupt* represents a class of events that are detected by the hardware or the system software. Interrupts are said to occur. An *occurrence* of an interrupt is separable into generation and delivery. *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. *Delivery* is the action that invokes part of the program as response to the interrupt occurrence. Between generation and delivery, the interrupt occurrence (or interrupt) is *pending*. Some or all interrupts may be *blocked*. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Certain interrupts are *reserved*. The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be *attached* to that interrupt. The execution of that program unit, the *interrupt handler*, is invoked upon delivery of the interrupt occurrence.

3 While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt. While the handler executes, the corresponding interrupt is blocked.

4 While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined.

5 Each interrupt has a *default treatment* which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is implementation defined, as is the method (if one exists) for configuring the default treatments for interrupts.

An interrupt is delivered to the handler (or default treatment) that is in effect for that interrupt at the time of delivery. 6

An exception propagated from a handler that is invoked by an interrupt has no effect. 7

If the Ceiling_Locking policy (see D.3) is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object. 8

Implementation Requirements

The implementation shall provide a mechanism to determine the minimum stack space that is needed for each interrupt handler and to reserve that space for the execution of the handler. This space should accommodate nested invocations of the handler where the system permits this. 9

If the hardware or the underlying system holds pending interrupt occurrences, the implementation shall provide for later delivery of these occurrences to the program. 10

If the Ceiling_Locking policy is not in effect, the implementation shall provide means for the application to specify whether interrupts are to be blocked during protected actions. 11

Documentation Requirements

The implementation shall document the following items: 12

1. For each interrupt, which interrupts are blocked from delivery when a handler attached to that interrupt executes (either as a result of an interrupt delivery or of an ordinary call on a procedure of the corresponding protected object). 13
2. Any interrupts that cannot be blocked, and the effect of attaching handlers to such interrupts, if this is permitted. 14
3. Which run-time stack an interrupt handler uses when it executes as a result of an interrupt delivery; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack. 15
4. Any implementation- or hardware-specific activity that happens before a user-defined interrupt handler gets control (e.g., reading device registers, acknowledging devices). 16
5. Any timing or other limitations imposed on the execution of interrupt handlers. 17
6. The state (blocked/unblocked) of the non-reserved interrupts when the program starts; if some interrupts are unblocked, what is the mechanism a program can use to protect itself before it can attach the corresponding handlers. 18
7. Whether the interrupted task is allowed to resume execution before the interrupt handler returns. 19
8. The treatment of interrupt occurrences that are generated while the interrupt is blocked; i.e., whether one or more occurrences are held for later delivery, or all are lost. 20
9. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt, and the mapping between the machine interrupts (or traps) and the predefined exceptions. 21
10. On a multi-processor, the rules governing the delivery of an interrupt to a particular processor. 22

Implementation Permissions

If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required as part of the execution of subprograms of a protected object whose one of its subprograms is an interrupt handler. 23

24 In a multi-processor with more than one interrupt subsystem, it is implementation defined whether (and how) interrupt sources from separate subsystems share the same Interrupt_ID type (see C.3.2). In particular, the meaning of a blocked or pending interrupt may then be applicable to one processor only.

25 Implementations are allowed to impose timing or other limitations on the execution of interrupt handlers.

26 Other forms of handlers are allowed to be supported, in which case, the rules of this subclause should be adhered to.

27 The active priority of the execution of an interrupt handler is allowed to vary from one occurrence of the same interrupt to another.

Implementation Advice

28 If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for a finer-grain control of interrupt blocking.

NOTES

29 1 The default treatment for an interrupt can be to keep the interrupt pending or to deliver it to an implementation-defined handler. Examples of actions that an implementation-defined handler is allowed to perform include aborting the partition, ignoring (i.e., discarding occurrences of) the interrupt, or queuing one or more occurrences of the interrupt for possible later delivery when a user-defined handler is attached to that interrupt.

30 2 It is a bounded error to call Task_Identification.Current_Task (see C.7.1) from an interrupt handler.

31 3 The rule that an exception propagated from an interrupt handler has no effect is modeled after the rule about exceptions propagated out of task bodies.

C.3.1 Protected Procedure Handlers

Syntax

1 The form of a pragma Interrupt_Handler is as follows:

2 **pragma** Interrupt_Handler(*handler*_name);

3 The form of a pragma Attach_Handler is as follows:

4 **pragma** Attach_Handler(*handler*_name, expression);

Name Resolution Rules

5 For the Interrupt_Handler and Attach_Handler pragmas, the *handler*_name shall resolve to denote a protected procedure with a parameterless profile.

6 For the Attach_Handler pragma, the expected type for the expression is Interrupts.Interrupt_ID (see C.3.2).

Legality Rules

7 The Attach_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library level declaration.

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

Dynamic Semantics

If the pragma Interrupt_Handler appears in a protected_definition, then the corresponding procedure can be attached dynamically, as a handler, to interrupts (see C.3.2). Such procedures are allowed to be attached to multiple interrupts. 9

The expression in the Attach_Handler pragma as evaluated at object creation time specifies an interrupt. As part of the initialization of that object, if the Attach_Handler pragma is specified, the *handler* procedure is attached to the specified interrupt. A check is made that the corresponding interrupt is not reserved. Program_Error is raised if the check fails, and the existing treatment for the interrupt is not affected. 10

If the Ceiling_Locking policy (see D.3) is in effect then upon the initialization of a protected object that either an Attach_Handler or Interrupt_Handler pragma applies to one of its procedures, a check is made that the ceiling priority defined in the protected_definition is in the range of System.Interrupt_Priority. If the check fails, Program_Error is raised. 11

When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. Otherwise, that is, if an Attach_Handler pragma was used, the previous handler is restored. 12

When a handler is attached to an interrupt, the interrupt is blocked (subject to the Implementation Permission in C.3) during the execution of every protected action on the protected object containing the handler. 13

Erroneous Execution

If the Ceiling_Locking policy (see D.3) is in effect and an interrupt is delivered to a handler, and the interrupt hardware priority is higher than the ceiling priority of the corresponding protected object, the execution of the program is erroneous. 14

Metrics

The following metric shall be documented by the implementation: 15

1. The worst case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C – (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt. 16

Implementation Permissions

When the pragmas Attach_Handler or Interrupt_Handler apply to a protected procedure, the implementation is allowed to impose implementation-defined restrictions on the corresponding protected_type_declaration and protected_body. 17

An implementation may use a different mechanism for invoking a protected procedure in response to a hardware interrupt than is used for a call to that protected procedure from a task. 18

Notwithstanding what this subclause says elsewhere, the Attach_Handler and Interrupt_Handler pragmas are allowed to be used for other, implementation defined, forms of interrupt handlers. 19

Implementation Advice

20 Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.

21 Whenever practical, the implementation should detect violations of any implementation-defined restrictions before run time.

NOTES

22 4 The Attach_Handler pragma can provide static attachment of handlers to interrupts if the implementation supports preelaboration of protected objects. (See C.4.)

23 5 The ceiling priority of a protected object that one of its procedures is attached to an interrupt should be at least as high as the highest processor priority at which that interrupt will ever be delivered.

24 6 Protected procedures can also be attached dynamically to interrupts via operations declared in the predefined package Interrupts.

25 7 An example of a possible implementation-defined restriction is disallowing the use of the standard storage pools within the body of a protected procedure that is an interrupt handler.

C.3.2 The Package Interrupts

Static Semantics

1 The following language-defined packages exist:

2
```
with System;
package Ada.Interrupts is
   type Interrupt_ID is implementation-defined;
   type Parameterless_Handler is
      access protected procedure;
```

3

4
```
   function Is_Reserved (Interrupt : Interrupt_ID)
      return Boolean;
```

5
```
   function Is_Attached (Interrupt : Interrupt_ID)
      return Boolean;
```

6
```
   function Current_Handler (Interrupt : Interrupt_ID)
      return Parameterless_Handler;
```

7
```
   procedure Attach_Handler
      (New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);
```

8
```
   procedure Exchange_Handler
      (Old_Handler : out Parameterless_Handler;
       New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);
```

9
```
   procedure Detach_Handler
      (Interrupt : in Interrupt_ID);
```

10
```
   function Reference(Interrupt : Interrupt_ID)
      return System.Address;
```

11
```
private
   ... -- not specified by the language
end Ada.Interrupts;
```

12

```
package Ada.Interrupts.Names is
   implementation-defined : constant Interrupt_ID :=
      implementation-defined;
      . . .
   implementation-defined : constant Interrupt_ID :=
      implementation-defined;
end Ada.Interrupts.Names;
```

Dynamic Semantics

The Interrupt_ID type is an implementation-defined discrete type used to identify interrupts. 13

The Is_Reserved function returns True if and only if the specified interrupt is reserved. 14

The Is_Attached function returns True if and only if a user-specified interrupt handler is attached to the interrupt. 15

The Current_Handler function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, Current_Handler returns a value that designates the default treatment; calling Attach_Handler or Exchange_Handler with this value restores the default treatment. 16

The Attach_Handler procedure attaches the specified handler to the interrupt, overriding any existing treatment (including a user handler) in effect for that interrupt. If New_Handler is **null**, the default treatment is restored. If New_Handler designates a protected procedure to which the pragma Interrupt_Handler does not apply, Program_Error is raised. In this case, the operation does not modify the existing interrupt treatment. 17

The Exchange_Handler procedure operates in the same manner as Attach_Handler with the addition that the value returned in Old_Handler designates the previous treatment for the specified interrupt. 18

The Detach_Handler procedure restores the default treatment for the specified interrupt. 19

For all operations defined in this package that take a parameter of type Interrupt_ID, with the exception of Is_Reserved and Reference, a check is made that the specified interrupt is not reserved. Program_Error is raised if this check fails. 20

If, by using the Attach_Handler, Detach_Handler, or Exchange_Handler procedures, an attempt is made to detach a handler that was attached statically (using the pragma Attach_Handler), the handler is not detached and Program_Error is raised. 21

The Reference function returns a value of type System.Address that can be used to attach a task entry, via an address clause (see J.7.1) to the interrupt specified by Interrupt. This function raises Program_Error if attaching task entries to interrupts (or to this particular interrupt) is not supported. 22

Implementation Requirements

At no time during attachment or exchange of handlers shall the current handler of the corresponding interrupt be undefined. 23

Documentation Requirements

If the Ceiling_Locking policy (see D.3) is in effect the implementation shall document the default ceiling priority assigned to a protected object that contains either the Attach_Handler or Interrupt_Handler pragmas, but not the Interrupt_Priority pragma. This default need not be the same for all interrupts. 24

Implementation Advice

If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts. 25

NOTES

26 8 The package Interrupts.Names contains implementation-defined names (and constant values) for the interrupts that are supported by the implementation.

Examples

27 *Example of interrupt handlers:*

28
```
Device_Priority : constant
  array (1..5) of System.Interrupt_Priority := ( ... );
protected type Device_Interface
  (Int_ID : Ada.Interrupts.Interrupt_ID) is
  procedure Handler;
  pragma Attach_Handler(Handler, Int_ID);
  ...
  pragma Interrupt_Priority(Device_Priority(Int_ID));
end Device_Interface;
  ...
Device_1_Driver : Device_Interface(1);
  ...
Device_5_Driver : Device_Interface(5);
  ...
```

C.4 Preelaboration Requirements

1 This clause specifies additional implementation and documentation requirements for the Preelaborate pragma (see 10.2.1).

Implementation Requirements

2 The implementation shall not incur any run-time overhead for the elaboration checks of subprograms and protected_bodies declared in preelaborated library units.

3 The implementation shall not execute any memory write operations after load time for the elaboration of constant objects declared immediately within the declarative region of a preelaborated library package, so long as the subtype and initial expression (or default initial expressions if initialized by default) of the object_declaration satisfy the following restrictions. The meaning of *load time* is implementation defined.

4 • Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any;

5 • any constraint is a static constraint;

6 • any allocator is for an access-to-constant type;

7 • any uses of predefined operators appear only within static expressions;

8 • any primaries that are names, other than attribute_references for the Access or Address attributes, appear only within static expressions;

9 • any name that is not part of a static expression is an expanded name or direct_name that statically denotes some entity;

10 • any discrete_choice of an array_aggregate is static;

11 • no language-defined check associated with the elaboration of the object_declaration can fail.

Documentation Requirements

12 The implementation shall document any circumstances under which the elaboration of a preelaborated package causes code to be executed at run time.

13 The implementation shall document whether the method used for initialization of preelaborated variables allows a partition to be restarted without reloading.

Implementation Advice

It is recommended that preelaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements. 14

C.5 Pragma Discard_Names

A pragma Discard_Names may be used to request a reduction in storage used for the names of certain entities. 1

Syntax

The form of a pragma Discard_Names is as follows: 2

pragma Discard_Names[([On =>] local_name)]; 3

A pragma Discard_Names is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma. 4

Legality Rules

The local_name (if present) shall denote a non-derived enumeration first subtype, a tagged first subtype, or an exception. The pragma applies to the type or exception. Without a local_name, the pragma applies to all such entities declared after the pragma, within the same declarative region. Alternatively, the pragma can be used as a configuration pragma. If the pragma applies to a type, then it applies also to all descendants of the type. 5

Static Semantics

If a local_name is given, then a pragma Discard_Names is a representation pragma. 6

If the pragma applies to an enumeration type, then the semantics of the Wide_Image and Wide_Value attributes are implementation defined for that type; the semantics of Image and Value are still defined in terms of Wide_Image and Wide_Value. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the pragma applies to a tagged type, then the semantics of the Tags.Expanded_Name function are implementation defined for that type. If the pragma applies to an exception, then the semantics of the Exceptions.Exception_Name function are implementation defined for that exception. 7

Implementation Advice

If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity. 8

C.6 Shared Variable Control

This clause specifies representation pragmas that control the use of shared variables. 1

Syntax

The form for pragmas Atomic, Volatile, Atomic_Components, and Volatile_Components is as follows: 2

pragma Atomic(local_name); 3

pragma Volatile(local_name); 4

pragma Atomic_Components(*array*_local_name); 5

pragma Volatile_Components(*array*_local_name); 6

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

8 A *volatile* type is one to which a pragma Volatile applies. A *volatile* object (including a component) is one to which a pragma Volatile applies, or a component of an array to which a pragma Volatile_Components applies, or any object of a volatile type. In addition, every atomic type or object is also defined to be volatile. Finally, if an object is volatile, then so are all of its subcomponents (the same does not apply to atomic).

Name Resolution Rules

9 The local_name in an Atomic or Volatile pragma shall resolve to denote either an object_declaration, a non-inherited component_declaration, or a full_type_declaration. The *array*_local_name in an Atomic_Components or Volatile_Components pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type.

Legality Rules

10 It is illegal to apply either an Atomic or Atomic_Components pragma to an object or type if the implementation cannot support the indivisible reads and updates required by the pragma (see below).

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

12 If an atomic object is passed as a parameter, then the type of the formal parameter shall either be atomic or allow pass by copy (that is, not be a nonatomic by-reference type). If an atomic object is used as an actual for a generic formal object of mode **in out**, then the type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object (including a component), then the designated type of the resulting access type shall be atomic. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic or allow pass by copy. Corresponding rules apply to volatile objects and types.

13 If a pragma Volatile, Volatile_Components, Atomic, or Atomic_Components applies to a stand-alone constant object, then a pragma Import shall also apply to it.

Static Semantics

14 These pragmas are representation pragmas (see 13.1).

Dynamic Semantics

15 For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible.

16 For a volatile object all reads and updates of the object as a whole are performed directly to memory.

17 Two actions are sequential (see 9.10) if each is the read or update of the same atomic object.

18 If a type is atomic or volatile and it is not a by-copy type, then the type is defined to be a by-reference type. If any subcomponent of a type is atomic or volatile, then the type is defined to be a by-reference type.

If an actual parameter is atomic or volatile, and the corresponding formal parameter is not, then the parameter is passed by copy. 19

Implementation Requirements

The external effect of a program (see 1.1.3) is defined to include each read and update of a volatile or atomic object. The implementation shall not generate any memory reads or updates of atomic or volatile objects other than those specified by the program. 20

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. 21

NOTES
9 An imported volatile or atomic constant behaves as a constant (i.e. read-only) with respect to other parts of the Ada program, but can still be modified by an ‘‘external source.’’ 22

C.7 Task Identification and Attributes

This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. 1

C.7.1 The Package Task_Identification

Static Semantics

The following language-defined library package exists: 1

```
package Ada.Task_Identification is                                    2
   type Task_ID is private;
   Null_Task_ID : constant Task_ID;
   function  "=" (Left, Right : Task_ID) return Boolean;
   function  Image       (T : Task_ID) return String;                 3
   function  Current_Task return Task_ID;
   procedure Abort_Task  (T : in out Task_ID);
   function  Is_Terminated(T : Task_ID) return Boolean;               4
   function  Is_Callable  (T : Task_ID) return Boolean;
private
   ... -- not specified by the language
end Ada.Task_Identification;
```

Dynamic Semantics

A value of the type Task_ID identifies an existent task. The constant Null_Task_ID does not identify any task. Each object of the type Task_ID is default initialized to the value of Null_Task_ID. 5

The function "=" returns True if and only if Left and Right identify the same task or both have the value Null_Task_ID. 6

The function Image returns an implementation-defined string that identifies T. If T equals Null_Task_ID, Image returns an empty string. 7

The function Current_Task returns a value that identifies the calling task. 8

The effect of Abort_Task is the same as the abort_statement for the task identified by T. In addition, if T identifies the environment task, the entire partition is aborted, See E.1. 9

The functions Is_Terminated and Is_Callable return the value of the corresponding attribute of the task identified by T. 10

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

12 T'Identity Yields a value of the type Task_ID that identifies the task denoted by T.

13 For a prefix E that denotes an entry_declaration, the following attribute is defined:

14 E'Caller Yields a value of the type Task_ID that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an entry_body or accept_statement corresponding to the entry_declaration denoted by E.

15 Program_Error is raised if a value of Null_Task_ID is passed as a parameter to Abort_Task, Is_Terminated, and Is_Callable.

16 Abort_Task is a potentially blocking operation (see 9.5.1).

Bounded (Run-Time) Errors

17 It is a bounded error to call the Current_Task function from an entry body or an interrupt handler. Program_Error is raised, or an implementation-defined value of the type Task_ID is returned.

Erroneous Execution

18 If a value of Task_ID is passed as a parameter to any of the operations declared in this package (or any language-defined child of this package), and the corresponding task object no longer exists, the execution of the program is erroneous.

Documentation Requirements

19 The implementation shall document the effect of calling Current_Task from an entry body or interrupt handler.

NOTES

20 10 This package is intended for use in writing user-defined task scheduling packages and constructing server tasks. Current_Task can be used in conjunction with other operations requiring a task as an argument such as Set_Priority (see D.5).

21 11 The function Current_Task and the attribute Caller can return a Task_ID value that identifies the environment task.

C.7.2 The Package Task_Attributes

Static Semantics

1 The following language-defined generic library package exists:

```
2    with Ada.Task_Identification; use Ada.Task_Identification;
     generic
        type Attribute is private;
        Initial_Value : in Attribute;
     package Ada.Task_Attributes is
3       type Attribute_Handle is access all Attribute;
4       function Value(T : Task_ID := Current_Task)
          return Attribute;
5       function Reference(T : Task_ID := Current_Task)
          return Attribute_Handle;
6       procedure Set_Value(Val : in Attribute;
                            T : in Task_ID := Current_Task);
        procedure Reinitialize(T : in Task_ID := Current_Task);
7    end Ada.Task_Attributes;
```

Dynamic Semantics

When an instance of Task_Attributes is elaborated in a given active partition, an object of the actual type corresponding to the formal type Attribute is implicitly created for each task (of that partition) that exists and is not yet terminated. This object acts as a user-defined attribute of the task. A task created previously in the partition and not yet terminated has this attribute from that point on. Each task subsequently created in the partition will have this attribute when created. In all these cases, the initial value of the given attribute is Initial_Value. 8

The Value operation returns the value of the corresponding attribute of T. 9

The Reference operation returns an access value that designates the corresponding attribute of T. 10

The Set_Value operation performs any finalization on the old value of the attribute of T and assigns Val to that attribute (see 5.2 and 7.6). 11

The effect of the Reinitialize operation is the same as Set_Value where the Val parameter is replaced with Initial_Value. 12

For all the operations declared in this package, Tasking_Error is raised if the task identified by T is terminated. Program_Error is raised if the value of T is Null_Task_ID. 13

Erroneous Execution

It is erroneous to dereference the access value returned by a given call on Reference after a subsequent call on Reinitialize for the same task attribute, or after the associated task terminates. 14

If a value of Task_ID is passed as a parameter to any of the operations declared in this package and the corresponding task object no longer exists, the execution of the program is erroneous. 15

Implementation Requirements

The implementation shall perform each of the above operations for a given attribute of a given task atomically with respect to any other of the above operations for the same attribute of the same task. 16

When a task terminates, the implementation shall finalize all attributes of the task, and reclaim any other storage associated with the attributes. 17

Documentation Requirements

The implementation shall document the limit on the number of attributes per task, if any, and the limit on the total storage for attribute values per task, if such a limit exists. 18

In addition, if these limits can be configured, the implementation shall document how to configure them. 19

Metrics

The implementation shall document the following metrics: A task calling the following subprograms shall execute in a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task T are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the Attribute type shall be a scalar whose size is equal to the size of the predefined integer size. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task (that is, the default value for the T parameter is used), and the other, where T identifies another, non-terminated, task. 20

21 The following calls (to subprograms in the Task_Attributes package) shall be measured:

22 • a call to Value, where the return value is Initial_Value;

23 • a call to Value, where the return value is not equal to Initial_Value;

24 • a call to Reference, where the return value designates a value equal to Initial_Value;

25 • a call to Reference, where the return value designates a value not equal to Initial_Value;

26 • a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value.

27 • a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is not equal to Initial_Value.

Implementation Permissions

28 An implementation need not actually create the object corresponding to a task attribute until its value is set to something other than that of Initial_Value, or until Reference is called for the task attribute. Similarly, when the value of the attribute is to be reinitialized to that of Initial_Value, the object may instead be finalized and its storage reclaimed, to be recreated when needed later. While the object does not exist, the function Value may simply return Initial_Value, rather than implicitly creating the object.

29 An implementation is allowed to place restrictions on the maximum number of attributes a task may have, the maximum size of each attribute, and the total storage size allocated for all the attributes of a task.

Implementation Advice

30 Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the task's attributes, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.

NOTES

31 12 An attribute always exists (after instantiation), and has the initial value. It need not occupy memory until the first operation that potentially changes the attribute value. The same holds true after Reinitialize.

32 13 The result of the Reference function should be used with care; it is always safe to use that result in the task body whose attribute is being accessed. However, when the result is being used by another task, the programmer must make sure that the task whose attribute is being accessed is not yet terminated. Failing to do so could make the program execution erroneous.

33 14 As specified in C.7.1, if the parameter T (in a call on a subprogram of an instance of this package) identifies a nonexistent task, the execution of the program is erroneous.

Annex D
(normative)

Real-Time Systems

This Annex specifies additional characteristics of Ada implementations intended for real-time systems software. To conform to this Annex, an implementation shall also conform to the Systems Programming Annex. 1

Metrics

The metrics are documentation requirements; an implementation shall document the values of the language-defined metrics for at least one configuration of hardware or an underlying system supported by the implementation, and shall document the details of that configuration. 2

The metrics do not necessarily yield a simple number. For some, a range is more suitable, for others a formula dependent on some parameter is appropriate, and for others, it may be more suitable to break the metric into several cases. Unless specified otherwise, the metrics in this annex are expressed in processor clock cycles. For metrics that require documentation of an upper bound, if there is no upper bound, the implementation shall report that the metric is unbounded. 3

NOTES

1 The specification of the metrics makes a distinction between upper bounds and simple execution times. Where something is just specified as ''the execution time of'' a piece of code, this leaves one the freedom to choose a nonpathological case. This kind of metric is of the form ''there exists a program such that the value of the metric is V''. Conversely, the meaning of upper bounds is ''there is no program such that the value of the metric is greater than V''. This kind of metric can only be partially tested, by finding the value of V for one or more test programs. 4

2 The metrics do not cover the whole language; they are limited to features that are specified in Annex C, ''Systems Programming'' and in this Annex. The metrics are intended to provide guidance to potential users as to whether a particular implementation of such a feature is going to be adequate for a particular real-time application. As such, the metrics are aimed at known implementation choices that can result in significant performance differences. 5

3 The purpose of the metrics is not necessarily to provide fine-grained quantitative results or to serve as a comparison between different implementations on the same or different platforms. Instead, their goal is rather qualitative; to define a standard set of approximate values that can be measured and used to estimate the general suitability of an implementation, or to evaluate the comparative utility of certain features of an implementation for a particular real-time application. 6

D.1 Task Priorities

This clause specifies the priority model for real-time systems. In addition, the methods for specifying priorities are defined. 1

Syntax

The form of a pragma Priority is as follows: 2

pragma Priority(expression); 3

The form of a pragma Interrupt_Priority is as follows: 4

pragma Interrupt_Priority[(expression)]; 5

Name Resolution Rules

6 The expected type for the expression in a Priority or Interrupt_Priority pragma is Integer.

Legality Rules

7 A Priority pragma is allowed only immediately within a task_definition, a protected_definition, or the declarative_part of a subprogram_body. An Interrupt_Priority pragma is allowed only immediately within a task_definition or a protected_definition. At most one such pragma shall appear within a given construct.

8 For a Priority pragma that appears in the declarative_part of a subprogram_body, the expression shall be static, and its value shall be in the range of System.Priority.

Static Semantics

9 The following declarations exist in package System:

10
```
subtype Any_Priority is Integer range implementation-defined;
subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last;
```

11
```
Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;
```

12 The full range of priority values supported by an implementation is specified by the subtype Any_Priority. The subrange of priority values that are high enough to require the blocking of one or more interrupts is specified by the subtype Interrupt_Priority. The subrange of priority values below System.Interrupt_Priority'First is specified by the subtype System.Priority.

13 The priority specified by a Priority or Interrupt_Priority pragma is the value of the expression in the pragma, if any. If there is no expression in an Interrupt_Priority pragma, the priority value is Interrupt_Priority'Last.

Dynamic Semantics

14 A Priority pragma has no effect if it occurs in the declarative_part of the subprogram_body of a subprogram other than the main subprogram.

15 A *task priority* is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The *base priority* of a task is the priority with which it was created, or to which it was later set by Dynamic_Priorities.Set_Priority (see D.5). At all times, a task also has an *active priority*, which generally reflects its base priority as well as any priority it inherits from other sources. *Priority inheritance* is the process by which the priority of a task or other entity (e.g. a protected object; see D.3) is used in the evaluation of another task's active priority.

16 The effect of specifying such a pragma in a protected_definition is discussed in D.3.

17 The expression in a Priority or Interrupt_Priority pragma that appears in a task_definition is evaluated for each task object (see 9.1). For a Priority pragma, the value of the expression is converted to the subtype Priority; for an Interrupt_Priority pragma, this value is converted to the subtype Any_Priority. The priority value is then associated with the task object whose task_definition contains the pragma.

18 Likewise, the priority value is associated with the environment task if the pragma appears in the declarative_part of the main subprogram.

The initial value of a task's base priority is specified by default or by means of a Priority or Interrupt_Priority pragma. After a task is created, its base priority can be changed only by a call to Dynamic_Priorities.Set_Priority (see D.5). The initial base priority of a task in the absence of a pragma is the base priority of the task that creates it at the time of creation (see 9.1). If a pragma Priority does not apply to the main subprogram, the initial base priority of the environment task is System.Default_Priority. The task's active priority is used when the task competes for processors. Similarly, the task's active priority is used to determine the task's position in any queue when Priority_Queuing is specified (see D.4). 19

At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see D.11), its base priority is always a source of priority inheritance. Other sources of priority inheritance are specified under the following conditions: 20

- During activation, a task being activated inherits the active priority of the its activator (see 9.2). 21
- During rendezvous, the task accepting the entry call inherits the active priority of the caller (see 9.5.3). 22
- During a protected action on a protected object, a task inherits the ceiling priority of the protected object (see 9.5 and D.3). 23

In all of these cases, the priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. 24

Implementation Requirements

The range of System.Interrupt_Priority shall include at least one value. 25

The range of System.Priority shall include at least 30 values. 26

NOTES

4 The priority expression can include references to discriminants of the enclosing type. 27

5 It is a consequence of the active priority rules that at the point when a task stops inheriting a priority from another source, its active priority is re-evaluated. This is in addition to other instances described in this Annex for such re-evaluation. 28

6 An implementation may provide a non-standard mode in which tasks inherit priorities under conditions other than those specified above. 29

D.2 Priority Scheduling

This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9.2). The rules have two parts: the task dispatching model (see D.2.1), and a specific task dispatching policy (see D.2.2). 1

D.2.1 The Task Dispatching Model

The task dispatching model specifies preemptive scheduling, based on conceptual priority-ordered ready queues. 1

Dynamic Semantics

A task runs (that is, it becomes a *running task*) only when it is ready (see 9.2) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority. 2

3 It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

4 *Task dispatching* is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called *task dispatching points*. A task reaches a task dispatching point whenever it becomes blocked, and whenever it becomes ready. In addition, the completion of an accept_statement (see 9.5.2), and task termination are task dispatching points for the executing task. Other task dispatching points are defined throughout this Annex.

5 *Task dispatching policies* are specified in terms of conceptual *ready queues*, task states, and task preemption. A ready queue is an ordered list of ready tasks. The first position in a queue is called the *head of the queue*, and the last position is called the *tail of the queue*. A task is *ready* if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

6 Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point, one task is selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

7 A preemptible resource is a resource that while allocated to one task can be allocated (temporarily) to another instead. Processors are preemptible resources. Access to a protected object (see 9.5.1) is a nonpreemptible resource. When a higher-priority task is dispatched to the processor, and the previously running task is placed on the appropriate ready queue, the latter task is said to be *preempted*.

8 A new running task is also selected whenever there is a nonempty ready queue with a higher priority than the priority of the running task, or when the task dispatching policy requires a running task to go back to a ready queue. These are also task dispatching points.

Implementation Permissions

9 An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation defined effect on task dispatching (see D.2.2).

10 An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range.

NOTES

11 7 Section 9 specifies under which circumstances a task becomes ready. The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. When a task is not ready, it is said to be blocked.

12 8 An example of a possible implementation-defined execution resource is a page of physical memory, which needs to be loaded with a particular page of virtual memory before a task can continue execution.

13 9 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.

14 10 While a task is running, it is not on any ready queue. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor.

15 11 In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as

sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.

12 The priority of a task is determined by rules specified in this subclause, and under D.1, ‘‘Task Priorities’’, D.3, ‘‘Priority Ceiling Locking’’, and D.5, ‘‘Dynamic Priorities’’. 16

D.2.2 The Standard Task Dispatching Policy

Syntax

The form of a pragma Task_Dispatching_Policy is as follows: 1

pragma Task_Dispatching_Policy(*policy*_identifier); 2

Legality Rules

The *policy*_identifier shall either be FIFO_Within_Priorities or an implementation-defined identifier. 3

Post-Compilation Rules

A Task_Dispatching_Policy pragma is a configuration pragma. 4

If the FIFO_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition. 5

Dynamic Semantics

A *task dispatching policy* specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues, and whether a task is inserted at the head or the tail of the queue for its active priority. The task dispatching policy is specified by a Task_Dispatching_Policy configuration pragma. If no such pragma appears in any of the program units comprising a partition, the task dispatching policy for that partition is unspecified. 6

The language defines only one task dispatching policy, FIFO_Within_Priorities; when this policy is in effect, modifications to the ready queues occur only as follows: 7

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority. 8
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority. 9
- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority. 10
- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority. 11

Each of the events specified above is a task dispatching point (see D.2.1). 12

In addition, when a task is preempted, it is added at the head of the ready queue for its active priority. 13

Documentation Requirements

Priority inversion is the duration for which a task remains at the head of the highest priority ready queue while the processor executes a lower priority task. The implementation shall document: 14

- The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and
- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

Implementation Permissions

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

For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation defined manner. However, a delay_statement always corresponds to at least one task dispatching point.

NOTES

13 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

14 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

15 Setting the base priority of a ready task causes the task to move to the end of the queue for its active priority, regardless of whether the active priority of the task actually changes.

D.3 Priority Ceiling Locking

This clause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the *ceiling priority* of a protected object.

Syntax

The form of a pragma Locking_Policy is as follows:

pragma Locking_Policy(*policy*_identifier);

Legality Rules

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

Post-Compilation Rules

A Locking_Policy pragma is a configuration pragma.

Dynamic Semantics

A locking policy specifies the details of protected object locking. These rules specify whether or not protected objects have priorities, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking. The *locking policy* is specified by a Locking_Policy pragma. For implementation-defined locking policies, the effect of a Priority or Interrupt_Priority pragma on a protected object is implementation defined. If no Locking_Policy pragma appears in any of the program units comprising a partition, the locking policy for that partition, as well as the effect of specifying either a Priority or Interrupt_Priority pragma for a protected object, are implementation defined.

There is one predefined locking policy, Ceiling_Locking; this policy is defined as follows:

- Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1. The ceiling priority of a protected object (or

ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

- The expression of a Priority or Interrupt_Priority pragma is evaluated as part of the creation of the corresponding protected object and converted to the subtype System.Any_Priority or System.Interrupt_Priority, respectively. The value of the expression is the ceiling priority of the corresponding protected object.
- If an Interrupt_Handler or Attach_Handler pragma (see C.3.1) appears in a protected_definition without an Interrupt_Priority pragma, the ceiling priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt_Priority.
- If no pragma Priority, Interrupt_Priority, Interrupt_Handler, or Attach_Handler is specified in the protected_definition, then the ceiling priority of the corresponding protected object is System.Priority'Last.
- While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.
- When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails.

Implementation Permissions

The implementation is allowed to round all ceilings in a certain subrange of System.Priority or System.Interrupt_Priority up to the top of that subrange, uniformly.

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

Since implementations are allowed to place restrictions on code that runs at an interrupt-level active priority (see C.3.1 and D.2.1), the implementation may implement a language feature in terms of a protected object with an implementation-defined ceiling, but the ceiling shall be no less than Priority'Last.

Implementation Advice

The implementation should use names that end with ''_Locking'' for implementation-defined locking policies.

NOTES

16 While a task executes in a protected action, it can be preempted only by tasks whose active priorities are higher than the ceiling priority of the protected object.

17 If a protected object has a ceiling priority in the range of Interrupt_Priority, certain interrupts are blocked while protected actions of that object execute. In the extreme, if the ceiling is Interrupt_Priority'Last, all blockable interrupts are blocked during that time.

18 The ceiling priority of a protected object has to be in the Interrupt_Priority range if one of its procedures is to be used as an interrupt handler (see C.3).

19 When specifying the ceiling of a protected object, one should choose a value that is at least as high as the highest active priority at which tasks can be executing when they call protected operations of that object. In determining this value the following factors, which can affect active priority, should be considered: the effect of Set_Priority, nested protected operations, entry calls, task activation, and other implementation-defined factors.

20 Attaching a protected procedure whose ceiling is below the interrupt hardware priority to an interrupt causes the execution of the program to be erroneous (see C.3.1).

23 21 On a single processor implementation, the ceiling priority rules guarantee that there is no possibility of deadlock involving only protected subprograms (excluding the case where a protected operation calls another protected operation on the same protected object).

D.4 Entry Queuing Policies

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

Syntax

2 The form of a pragma Queuing_Policy is as follows:

3 **pragma** Queuing_Policy(*policy*_identifier);

Legality Rules

4 The *policy*_identifier shall be either FIFO_Queuing, Priority_Queuing or an implementation-defined identifier.

Post-Compilation Rules

5 A Queuing_Policy pragma is a configuration pragma.

Dynamic Semantics

6 A *queuing policy* governs the order in which tasks are queued for entry service, and the order in which different entry queues are considered for service. The queuing policy is specified by a Queuing_Policy pragma.

7 Two queuing policies, FIFO_Queuing and Priority_Queuing, are language defined. If no Queuing_Policy pragma appears in any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for this policy are specified in 9.5.3 and 9.7.1.

8 The Priority_Queuing policy is defined as follows:

9 • The calls to an entry (including a member of an entry family) are queued in an order consistent with the priorities of the calls. The *priority of an entry call* is initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order).

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

11 • When the base priority of a task is set (see D.5), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority.

12 • When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call with the highest priority is selected. If more than one such call has the same priority, the call that is queued on the entry whose declaration is first in textual order in the protected_definition is selected. For members of the same entry family, the one with the lower family index is selected.

13 • If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed.

- When more than one alternative of a selective_accept is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the accept_alternative that is first in textual order in the selective_accept is selected. 14

Implementation Permissions

Implementations are allowed to define other queuing policies, but need not support more than one such policy per partition. 15

Implementation Advice

The implementation should use names that end with ‘‘_Queuing’’ for implementation-defined queuing policies. 16

D.5 Dynamic Priorities

This clause specifies how the base priority of a task can be modified or queried at run time. 1

Static Semantics

The following language-defined library package exists: 2

```
with System;                                                                    3
with Ada.Task_Identification; -- See C.7.1
package Ada.Dynamic_Priorities is
    procedure Set_Priority(Priority : in System.Any_Priority;                   4
                           T : in Ada.Task_Identification.Task_ID :=
                           Ada.Task_Identification.Current_Task);
    function Get_Priority (T : Ada.Task_Identification.Task_ID :=               5
                           Ada.Task_Identification.Current_Task)
                           return System.Any_Priority;
end Ada.Dynamic_Priorities;                                                     6
```

Dynamic Semantics

The procedure Set_Priority sets the base priority of the specified task to the specified Priority value. Set_Priority has no effect if the task is terminated. 7

The function Get_Priority returns T’s current base priority. Tasking_Error is raised if the task is terminated. 8

Program_Error is raised by Set_Priority and Get_Priority if T is equal to Null_Task_ID. 9

Setting the task’s base priority to the new value takes place as soon as is practical but not while the task is performing a protected action. This setting occurs no later then the next abort completion point of the task T (see 9.8). 10

Bounded (Run-Time) Errors

If a task is blocked on a protected entry call, and the call is queued, it is a bounded error to raise its base priority above the ceiling priority of the corresponding protected object. When an entry call is cancelled, it is a bounded error if the priority of the calling task is higher than the ceiling priority of the corresponding protected object. In either of these cases, either Program_Error is raised in the task that called the entry, or its priority is temporarily lowered, or both, or neither. 11

Erroneous Execution

If any subprogram in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous. 12

Metrics

13 The implementation shall document the following metric:

14 • The execution time of a call to Set_Priority, for the nonpreempting case, in processor clock cycles. This is measured for a call that modifies the priority of a ready task that is not running (which cannot be the calling one), where the new base priority of the affected task is lower than the active priority of the calling task, and the affected task is not on any entry queue and is not executing a protected operation.

NOTES

15 22 Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the standard task dispatching policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged.

16 23 Under the priority queuing policy, setting a task's base priority has an effect on a queued entry call if the task is blocked waiting for the call. That is, setting the base priority of a task causes the priority of a queued entry call from that task to be updated and the call to be removed and then reinserted in the entry queue at the new priority (see D.4), unless the call originated from the triggering_statement of an asynchronous_select.

17 24 The effect of two or more Set_Priority calls executed in parallel on the same task is defined as executing these calls in some serial order.

18 25 The rule for when Tasking_Error is raised for Set_Priority or Get_Priority is different from the rule for when Tasking_Error is raised on an entry call (see 9.5.3). In particular, setting or querying the priority of a completed or an abnormal task is allowed, so long as the task is not yet terminated.

19 26 Changing the priorities of a set of tasks can be performed by a series of calls to Set_Priority for each task separately. For this to work reliably, it should be done within a protected operation that has high enough ceiling priority to guarantee that the operation completes without being preempted by any of the affected tasks.

D.6 Preemptive Abort

1 This clause specifies requirements on the immediacy with which an aborted construct is completed.

Dynamic Semantics

2 On a system with a single processor, an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation.

Documentation Requirements

3 On a multiprocessor, the implementation shall document any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor.

Metrics

4 The implementation shall document the following metrics:

5 • The execution time, in processor clock cycles, that it takes for an abort_statement to cause the completion of the aborted task. This is measured in a situation where a task T2 preempts task T1 and aborts T1. T1 does not have any finalization code. T2 shall verify that T1 has terminated, by means of the Terminated attribute.

6 • On a multiprocessor, an upper bound in seconds, on the time that the completion of an aborted task can be delayed beyond the point that it is required for a single processor.

7 • An upper bound on the execution time of an asynchronous_select, in processor clock cycles. This is measured between a point immediately before a task T1 executes a protected operation Pr.Set that makes the condition of an entry_barrier Pr.Wait true, and the point where task T2 resumes execution immediately after an entry call to Pr.Wait in an asynchronous_select. T1 preempts T2 while T2 is executing the abortable part, and then blocks itself so that T2 can execute. The execution time of T1 is measured separately, and subtracted.

- An upper bound on the execution time of an asynchronous_select, in the case that no asynchronous transfer of control takes place. This is measured between a point immediately before a task executes the asynchronous_select with a nonnull abortable part, and the point where the task continues execution immediately after it. The execution time of the abortable part is subtracted.

Implementation Advice

Even though the abort_statement is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the abort_statement to block.

On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this.

NOTES

27 Abortion does not change the active or base priority of the aborted task.

28 Abortion cannot be more immediate than is allowed by the rules for deferral of abortion during finalization and in protected actions.

D.7 Tasking Restrictions

This clause defines restrictions that can be used with a pragma Restrictions (see 13.12) to facilitate the construction of highly efficient tasking run-time systems.

Static Semantics

The following *restriction_*identifiers are language defined:

No_Task_Hierarchy
: All (nonenvironment) tasks depend directly on the environment task of the partition.

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

No_Abort_Statements
: There are no abort_statements, and there are no calls on Task_Identification.Abort_Task.

No_Terminate_Alternatives
: There are no selective_accepts with terminate_alternatives.

No_Task_Allocators
: There are no allocators for task types or types containing task subcomponents.

No_Implicit_Heap_Allocations
: There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.

No_Dynamic_Priorities
: There are no semantic dependences on the package Dynamic_Priorities.

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

The following *restriction_parameter_*identifiers are language defined:

Max_Select_Alternatives
: Specifies the maximum number of alternatives in a selective_accept.

13 Max_Task_Entries Specifies the maximum number of entries per task. The bounds of every entry family of a task unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. A value of zero indicates that no rendezvous are possible.

14 Max_Protected_Entries
Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.

Dynamic Semantics

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

16 The following *restriction_parameter*_identifiers are language defined:

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

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

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

20 It is implementation defined whether the use of pragma Restrictions results in a reduction in executable program size, storage requirements, or execution time. If possible, the implementation should provide quantitative descriptions of such effects for each restriction.

Implementation Advice

21 When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation.

NOTES

22 29 The above Storage_Checks can be suppressed with pragma Suppress.

D.8 Monotonic Time

1 This clause specifies a high-resolution, monotonic clock package.

Static Semantics

2 The following language-defined library package exists:

```
3     package Ada.Real_Time is
4       type Time is private;
        Time_First : constant Time;
        Time_Last : constant Time;
        Time_Unit : constant := implementation-defined-real-number;
5
6       type Time_Span is private;
        Time_Span_First : constant Time_Span;
        Time_Span_Last : constant Time_Span;
        Time_Span_Zero : constant Time_Span;
        Time_Span_Unit : constant Time_Span;
7
```

```
   Tick : constant Time_Span;
   function Clock return Time;

   function "+" (Left : Time; Right : Time_Span) return Time;
   function "+" (Left : Time_Span; Right : Time) return Time;
   function "-" (Left : Time; Right : Time_Span) return Time;
   function "-" (Left : Time; Right : Time) return Time_Span;

   function "<" (Left, Right : Time) return Boolean;
   function "<="(Left, Right : Time) return Boolean;
   function ">" (Left, Right : Time) return Boolean;
   function ">="(Left, Right : Time) return Boolean;

   function "+" (Left, Right : Time_Span) return Time_Span;
   function "-" (Left, Right : Time_Span) return Time_Span;
   function "-" (Right : Time_Span) return Time_Span;
   function "*" (Left : Time_Span; Right : Integer) return Time_Span;
   function "*" (Left : Integer; Right : Time_Span) return Time_Span;
   function "/" (Left, Right : Time_Span) return Integer;
   function "/" (Left : Time_Span; Right : Integer) return Time_Span;
   function "abs"(Right : Time_Span) return Time_Span;

   function "<" (Left, Right : Time_Span) return Boolean;
   function "<="(Left, Right : Time_Span) return Boolean;
   function ">" (Left, Right : Time_Span) return Boolean;
   function ">="(Left, Right : Time_Span) return Boolean;

   function To_Duration (TS : Time_Span) return Duration;
   function To_Time_Span (D : Duration) return Time_Span;

   function Nanoseconds  (NS : Integer) return Time_Span;
   function Microseconds (US : Integer) return Time_Span;
   function Milliseconds (MS : Integer) return Time_Span;
   type Seconds_Count is range implementation-defined;
   procedure Split(T : in Time; SC : out Seconds_Count; TS : out Time_Span);
   function Time_Of(SC : Seconds_Count; TS : Time_Span) return Time;
private
   ... -- not specified by the language
end Ada.Real_Time;
```

In this Annex, *real time* is defined to be the physical time as observed in the external environment. The type Time is a *time type* as defined by 9.6; values of this type may be used in a delay_until_statement. Values of this type represent segments of an ideal time line. The set of values of the type Time corresponds one-to-one with an implementation-defined range of mathematical integers.

The Time value I represents the half-open real time interval that starts with E+I*Time_Unit and is limited by E+(I+1)*Time_Unit, where Time_Unit is an implementation-defined real number and E is an unspecified origin point, the *epoch*, that is the same for all values of the type Time. It is not specified by the language whether the time values are synchronized with any standard time reference. For example, E can correspond to the time of system initialization or it can correspond to the epoch of some time standard.

Values of the type Time_Span represent length of real time duration. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers. The Time_Span value corresponding to the integer I represents the real-time duration I*Time_Unit.

21 Time_First and Time_Last are the smallest and largest values of the Time type, respectively. Similarly, Time_Span_First and Time_Span_Last are the smallest and largest values of the Time_Span type, respectively.

22 A value of type Seconds_Count represents an elapsed time, measured in seconds, since the epoch.

Dynamic Semantics

23 Time_Unit is the smallest amount of real time representable by the Time type; it is expressed in seconds. Time_Span_Unit is the difference between two successive values of the Time type. It is also the smallest positive value of type Time_Span. Time_Unit and Time_Span_Unit represent the same real time duration. A *clock tick* is a real time interval during which the clock value (as observed by calling the Clock function) remains constant. Tick is the average length of such intervals.

24 The function To_Duration converts the value TS to a value of type Duration. Similarly, the function To_Time_Span converts the value D to a value of type Time_Span. For both operations, the result is rounded to the nearest exactly representable value (away from zero if exactly halfway between two exactly representable values).

25 To_Duration(Time_Span_Zero) returns 0.0, and To_Time_Span(0.0) returns Time_Span_Zero.

26 The functions Nanoseconds, Microseconds, and Milliseconds convert the input parameter to a value of the type Time_Span. NS, US, and MS are interpreted as a number of nanoseconds, microseconds, and milliseconds respectively. The result is rounded to the nearest exactly representable value (away from zero if exactly halfway between two exactly representable values).

27 The effects of the operators on Time and Time_Span are as for the operators defined for integer types.

28 The function Clock returns the amount of time since the epoch.

29 The effects of the Split and Time_Of operations are defined as follows, treating values of type Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split(T,SC,TS) is to set SC and TS to values such that T*Time_Unit = SC*1.0 + TS*Time_Unit, and 0.0 <= TS*Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the value T such that T*Time_Unit = SC*1.0 + TS*Time_Unit.

Implementation Requirements

30 The range of Time values shall be sufficient to uniquely represent the range of real times from program start-up to 50 years later. Tick shall be no greater than 1 millisecond. Time_Unit shall be less than or equal to 20 microseconds.

31 Time_Span_First shall be no greater than –3600 seconds, and Time_Span_Last shall be no less than 3600 seconds.

32 A *clock jump* is the difference between two successive distinct values of the clock (as observed by calling the Clock function). There shall be no backward clock jumps.

Documentation Requirements

33 The implementation shall document the values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick.

34 The implementation shall document the properties of the underlying time base used for the clock and for type Time, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied. 35

The implementation shall document any aspects of the the external environment that could interfere with the clock behavior as defined in this clause. 36

Metrics

For the purpose of the metrics defined in this clause, real time is defined to be the International Atomic Time (TAI). 37

The implementation shall document the following metrics: 38

- An upper bound on the real-time duration of a clock tick. This is a value D such that if t1 and t2 are any real times such that t1 < t2 and $Clock_{t1} = Clock_{t2}$ then t2 – t1 <= D. 39
- An upper bound on the size of a clock jump. 40
- An upper bound on the *drift rate* of Clock with respect to real time. This is a real number D such that 41

  $E*(1-D) <= (Clock_{t+E} - Clock_t) <= E*(1+D)$
  provided that: $Clock_t + E*(1+D) <= Time\_Last$. 42
- where $Clock_t$ is the value of Clock at time t, and E is a real time duration not less than 24 hours. The value of E used for this metric shall be reported. 43
- An upper bound on the execution time of a call to the Clock function, in processor clock cycles. 44
- Upper bounds on the execution times of the operators of the types Time and Time_Span, in processor clock cycles. 45

Implementation Permissions

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the Time and Time_Span types. 46

Implementation Advice

When appropriate, implementations should provide configuration mechanisms to change the value of Tick. 47

It is recommended that Calendar.Clock and Real_Time.Clock be implemented as transformations of the same time base. 48

It is recommended that the ‘‘best’’ time base which exists in the underlying system be available to the application through Clock. ‘‘Best’’ may mean highest accuracy or largest range. 49

NOTES

30 The rules in this clause do not imply that the implementation can protect the user from operator or installation errors which could result in the clock being set incorrectly. 50

31 Time_Unit is the granularity of the Time type. In contrast, Tick represents the granularity of Real_Time.Clock. There is no requirement that these be the same. 51

D.9 Delay Accuracy

1 This clause specifies performance requirements for the delay_statement. The rules apply both to delay_relative_statement and to delay_until_statement. Similarly, they apply equally to a simple delay_statement and to one which appears in a delay_alternative.

Dynamic Semantics

2 The effect of the delay_statement for Real_Time.Time is defined in terms of Real_Time.Clock:

3 • If C_1 is a value of Clock read before a task executes a delay_relative_statement with duration D, and C_2 is a value of Clock read after the task resumes execution following that delay_statement, then $C_2 - C_1 >= D$.

4 • If C is a value of Clock read after a task resumes execution following a delay_until_statement with Real_Time.Time value T, then C >= T.

5 A simple delay_statement with a negative or zero value for the expiration time does not cause the calling task to be blocked; it is nevertheless a potentially blocking operation (see 9.5.1).

6 When a delay_statement appears in a delay_alternative of a timed_entry_call the selection of the entry call is attempted, regardless of the specified expiration time. When a delay_statement appears in a selective_accept_alternative, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression.

Documentation Requirements

7 The implementation shall document the minimum value of the delay expression of a delay_relative_statement that causes the task to actually be blocked.

8 The implementation shall document the minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked.

Metrics

9 The implementation shall document the following metrics:

10 • An upper bound on the execution time, in processor clock cycles, of a delay_relative_statement whose requested value of the delay expression is less than or equal to zero.

11 • An upper bound on the execution time, in processor clock cycles, of a delay_until_statement whose requested value of the delay expression is less than or equal to the value of Real_Time.Clock at the time of executing the statement. Similarly, for Calendar.Clock.

12 • An upper bound on the *lateness* of a delay_relative_statement, for a positive value of the delay expression, in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready, and does not need to wait for any other execution resources. The upper bound is expressed as a function of the value of the delay expression. The lateness is obtained by subtracting the value of the delay expression from the *actual duration*. The actual duration is measured from a point immediately before a task executes the delay_statement to a point immediately after the task resumes execution following this statement.

13 • An upper bound on the lateness of a delay_until_statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a delay_until_statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement.

NOTES
32 The execution time of a delay_statement that does not cause the task to be blocked (e.g. ‘‘**delay** 0.0;’’) is of interest in situations where delays are used to achieve voluntary round-robin task dispatching among equal-priority tasks. 14

D.10 Synchronous Task Control

This clause describes a language-defined private semaphore (suspension object), which can be used for *two-stage suspend* operations and as a simple building block for implementing higher-level queues. 1

Static Semantics

The following language-defined package exists: 2

```
package Ada.Synchronous_Task_Control is                                    3
  type Suspension_Object is limited private;                               4
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
     ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The type Suspension_Object is a by-reference type. 5

Dynamic Semantics

An object of the type Suspension_Object has two visible states: true and false. Upon initialization, its value is set to false. 6

The operations Set_True and Set_False are atomic with respect to each other and with respect to Suspend_Until_True; they set the state to true and false respectively. 7

Current_State returns the current state of the object. 8

The procedure Suspend_Until_True blocks the calling task until the state of the object S is true; at that point the task becomes ready and the state of the object becomes false. 9

Program_Error is raised upon calling Suspend_Until_True if another task is already waiting on that suspension object. Suspend_Until_True is a potentially blocking operation (see 9.5.1). 10

Implementation Requirements

The implementation is required to allow the calling of Set_False and Set_True during any protected action, even one that has its ceiling priority in the Interrupt_Priority range. 11

D.11 Asynchronous Task Control

This clause introduces a language-defined package to do asynchronous suspend/resume on tasks. It uses a conceptual *held priority* value to represent the task’s *held* state. 1

Static Semantics

The following language-defined library package exists: 2

```
with Ada.Task_Identification;                                              3
package Ada.Asynchronous_Task_Control is
  procedure Hold(T : in Ada.Task_Identification.Task_ID);
  procedure Continue(T : in Ada.Task_Identification.Task_ID);
  function Is_Held(T : Ada.Task_Identification.Task_ID)
   return Boolean;
end Ada.Asynchronous_Task_Control;
```

Dynamic Semantics

After the Hold operation has been applied to a task, the task becomes *held*. For each processor there is a conceptual *idle task*, which is always ready. The base priority of the idle task is below System.Any_Priority'First. The *held priority* is a constant of the type integer whose value is below the base priority of the idle task.

The Hold operation sets the state of T to held. For a held task: the task's own base priority does not constitute an inheritance source (see D.1), and the value of the held priority is defined to be such a source instead.

The Continue operation resets the state of T to not-held; T's active priority is then reevaluated as described in D.1. This time, T's base priority is taken into account.

The Is_Held function returns True if and only if T is in the held state.

As part of these operations, a check is made that the task identified by T is not terminated. Tasking_Error is raised if the check fails. Program_Error is raised if the value of T is Null_Task_ID.

Erroneous Execution

If any operation in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous.

Implementation Permissions

An implementation need not support Asynchronous_Task_Control if it is infeasible to support it in the target environment.

NOTES

33 It is a consequence of the priority rules that held tasks cannot be dispatched on any processor in a partition (unless they are inheriting priorities) since their priorities are defined to be below the priority of any idle task.

34 The effect of calling Get_Priority and Set_Priority on a Held task is the same as on any other task.

35 Calling Hold on a held task or Continue on a non-held task has no effect.

36 The rules affecting queuing are derived from the above rules, in addition to the normal priority rules:

- When a held task is on the ready queue, its priority is so low as to never reach the top of the queue as long as there are other tasks on that queue.
- If a task is executing in a protected action, inside a rendezvous, or is inheriting priorities from other sources (e.g. when activated), it continues to execute until it is no longer executing the corresponding construct.
- If a task becomes held while waiting (as a caller) for a rendezvous to complete, the active priority of the accepting task is not affected.
- If a task becomes held while waiting in a selective_accept, and a entry call is issued to one of the open entries, the corresponding accept body executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another Continue.
- The same holds if the held task is the only task on a protected entry queue whose barrier becomes open. The corresponding entry body executes.

D.12 Other Optimizations and Determinism Rules

This clause describes various requirements for improving the response and determinism in a real-time system.

Implementation Requirements

If the implementation blocks interrupts (see C.3) not as a result of direct user action (e.g. an execution of a protected action) there shall be an upper bound on the duration of this blocking.

The implementation shall recognize entry-less protected types. The overhead of acquiring the execution resource of an object of such a type (see 9.5.1) shall be minimized. In particular, there should not be any overhead due to evaluating entry_barrier conditions.

Unchecked_Deallocation shall be supported for terminated tasks that are designated by access types, and shall have the effect of releasing all the storage associated with the task. This includes any run-time system or heap storage that has been implicitly allocated for the task by the implementation.

Documentation Requirements

The implementation shall document the upper bound on the duration of interrupt blocking caused by the implementation. If this is different for different interrupts or interrupt priority levels, it should be documented for each case.

Metrics

The implementation shall document the following metric:

- The overhead associated with obtaining a mutual-exclusive access to an entry-less protected object. This shall be measured in the following way:

  For a protected object of the form:

```
protected Lock is
   procedure Set;
   function Read return Boolean;
private
   Flag : Boolean := False;
end Lock;
protected body Lock is
   procedure Set is
   begin
      Flag := True;
   end Set;
   function Read return Boolean
   Begin
      return Flag;
   end Read;
end Lock;
```

  The execution time, in processor clock cycles, of a call to Set. This shall be measured between the point just before issuing the call, and the point just after the call completes. The function Read shall be called later to verify that Set was indeed called (and not optimized away). The calling task shall have sufficiently high priority as to not be preempted during the measurement period. The protected object shall have sufficiently high ceiling priority to allow the task to call Set.

  For a multiprocessor, if supported, the metric shall be reported for the case where no contention (on the execution resource) exists from tasks executing on other processors.