Rationale for Ada 2005: 4 Tasking and Real-Time

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements in the tasking and real-time areas for Ada 2005.

There are only a few changes to the core tasking model itself. One major extension, however, is the ability to combine the interface feature described in an earlier paper with the tasking model; this draws together the object-oriented and tasking models of Ada which previously were disjoint aspects of the language.

There are also many additional predefined packages in the Real-Time Systems annex concerning matters such as scheduling and timing; these form the major topic of this paper.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 3, September 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies real-time systems as an important area. It says

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

It then identifies the inclusion of the Ravenscar profile [2] (for predictable real-time) as a worthwhile addition and then asks the ARG to pay particular attention to

> Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns. Within this area it cites as high priority, improvements in the real-time features and improvements in the high integrity features.

Ada 2005 does indeed make many improvements in the real-time area and includes the Ravenscar profile as specifically mentioned. The following Ada issues cover the relevant changes and are described in detail in this paper:

249 Ravenscar profile for high-integrity systems

265 Partition elaboration policy for high-integrity systems

266 Task termination procedure

297 Timing events

298 Non-preemptive dispatching

305 New pragma and restrictions for real-time systems

307 Execution-time clocks

321 Definition of dispatching policies

327 Dynamic ceiling priorities

345 Protected and task interfaces

347 Title of Annex H

354 Group execution-time budgets

355 Priority dispatching including Round Robin

357 Earliest Deadline First scheduling

386 Further functions returning time-span values

394 Redundant Restrictions identifiers and Ravenscar

397 Conformance and overriding for procedures and entries

399 Single tasks and protected objects with interfaces

421 Sequential activation and attachment

These changes can be grouped as follows.

First there is the introduction of a mechanism for monitoring task termination (266).

A major innovation in the core language is the introduction of synchronized interfaces which provide a high degree of unification between the object-oriented and real-time aspects of Ada (345, 397, 399).

There is of course the introduction of the Ravenscar profile (249) plus associated restrictions (305, 394) in the Real-Time Systems annex (D).

There are major improvement to the scheduling and task dispatching mechanisms with the addition of further standard policies (298, 321, 327, 355, 357). These are also in Annex D.

A number of timing mechanisms are now provided. These concern stand-alone timers, timers for monitoring the CPU time of a single task, and timers for controlling the budgeting of time for groups of tasks (297, 307, 354, 386). Again these are in Annex D.

Finally, more control is provided over partition elaboration which is very relevant to real-time high-integrity systems (265, 421). This is in Annex H which is now entitled High Integrity Systems (347).

Note that further operations for the manipulation of time in child packages of Calendar (351) will be discussed with the predefined library in a later paper.

2 Task termination

In the Introduction we mentioned the problem of how tasks can have a silent death in Ada 95. This happens if a task raises an exception which is not handled by the task itself. Tasks may also terminate because of going abnormal as well as terminating normally. The detection of task termination and its causes can be monitored in Ada 2005 by the package Ada.Task_Termination whose specification is essentially

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Exceptions; use Ada.Exceptions;
package Ada.Task_Termination is
  pragma Preelaborable(Task_Termination);

  type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);
```

```
   type Termination_Handler is access protected
      procedure(Cause: in Cause_Of_Termination;
                  T: in Task_Id; X: in Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler (Handler: in Termination_Handler);
   function Current_Task_Fallback_Handler return Termination_Handler;

   procedure Set_Specific_Handler(T: in Task_Id; Handler: in Termination_Handler);
   function Specific_Handler(T: in Task_Id) return Termination_Handler;

end Ada.Task_Termination;
```

(Note that the above includes use clauses in order to simplify the presentation; the actual package does not have use clauses. We will use a similar approach for the other predefined packages described in this paper.)

The general idea is that we can associate a protected procedure with a task. The protected procedure is then invoked when the task terminates with an indication of the reason passed via its parameters. The protected procedure is identified by using the type Termination_Handler which is an access type referring to a protected procedure.

The association can be done in two ways. Thus (as in the Introduction) we might declare a protected object Grim_Reaper

```
protected Grim_Reaper is
  procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence);
end Grim_Reaper;
```

which contains the protected procedure Last_Gasp. Note that the parameters of Last_Gasp match those of the access type Termination_Handler.

We can then nominate Last_Gasp as the protected procedure to be called when the specific task T dies by

```
Set_Specific_Handler(T'Identity, Grim_Reaper.Last_Gasp'Access);
```

Alternatively we can nominate Last_Gasp as the protected procedure to be called when any of the tasks dependent on the current task becomes terminated by writing

```
Set_Dependents_Fallback_Handler(Grim_Reaper.Last_Gasp'Access);
```

Note that a task is not dependent upon itself and so this does not set a handler for the current task.

Thus a task can have two handlers. A fallback handler and a specific handler and either or both of these can be null. When a task terminates (that is after any finalization but just before it vanishes), the specific handler is invoked if it is not null. If the specific handler is null, then the fallback handler is invoked unless it too is null. If both are null then no handler is invoked.

The body of protected procedure Last_Gasp might then output various diagnostic messages

```
procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence) is
begin
  case C is
    when Normal => null;
    when Abnormal =>
      Put("Something nasty happened to task ");
      Put_Line(Image(T));
    when Unhandled_Exception =>
      Put("Unhandled exception occurred in task ");
      Put_Line(Image(T));
```

```
      Put(Exception_Information(X));
  end case;
end Last_Gasp;
```

There are three possible reasons for termination, it could be normal, abnormal (caused by abort), or because of propagation of an unhandled exception. In the last case the parameter X gives details of the exception occurrence whereas in the other cases X has the value Null_Occurrence.

Initially both specific and fallback handlers are null for all tasks. However, note that if a fallback handler has been set for all dependent tasks of T then the handler will also apply to any task subsequently created by T or one of its descendants. Thus a task can be born with a fallback handler already in place.

If a new handler is set then it replaces any existing handler of the appropriate kind. Calling either setting procedure with null for the handler naturally sets the appropriate handler to null.

The current handlers can be found by calling the functions Current_Task_Fallback_Handler or Specific_Handler; they return null if the handler is null.

It is important to realise that the fallback handlers for the tasks dependent on T need not all be the same since one of the dependent tasks of T might set a different handler for its own dependent tasks. Thus the fallback handlers for a tree of tasks can be different in various subtrees. This structure is reflected by the fact that the determination of the current fallback handler of a task is in fact done by searching recursively the tasks on which it depends.

Note that we cannot directly interrogate the fallback handler of a specific task but only that of the current task. Moreover, if a task sets a fallback handler for its dependents and then enquires of its own fallback handler it will not in general get the same answer because it is not one of its own dependents.

It is important to understand the situation regarding the environment task. This unnamed task is the task that elaborates the library units and then calls the main subprogram. Remember that library tasks (that is tasks declared at library level) are activated by the environment task before it calls the main subprogram.

Suppose the main subprogram calls the setting procedures as follows

```
procedure Main is

  protected RIP is
    protected procedure One( ... );
    protected procedure Two( ... );
  end;
  ...
begin
  Set_Dependents_Fallback_Handler(RIP.One'Access);
  Set_Specific_Handler(Current_Task, RIP.Two'Access);
  ...
end Main;
```

The specific handler for the environment task is then set to Two (because Current_Task is the environment task at this point) but the fallback handler for the environment task is null. On the other hand the fallback handler for all other tasks in the program including any library tasks is set to One. Note that it is not possible to set the fallback handler for the environment task.

The astute reader will note that there is actually a race condition here since a library task might have terminated before the handler gets set. We could overcome this by setting the handler as part of the elaboration code thus

```
package Start_Up is
  pragma Elaborate_Body;
end;

with Ada.Task_Termination; use Ada.Task_Termination;
package body Start_Up is
begin
  Set_Dependents_Fallback_Handler(RIP.One'Access);
end Start_Up;

with Start_Up;
pragma Elaborate(Start_Up);
package Library_Tasks is
  ...                                   -- declare library tasks here
end;
```

Note how the use of pragmas Elaborate_Body and Elaborate ensures that things get done in the correct order.

Some minor points are that if we try to set the specific handler for a task that has already terminated then Tasking_Error is raised. And if we try to set the specific handler for the null task, that is call Set_Specific_Handler with parameter T equal to Null_Task_Id, then Program_Error is raised. These exceptions are also raised by calls of the function Specific_Handler in similar circumstances.

3 Synchronized interfaces

We now turn to the most important improvement to the core tasking features introduced by Ada 2005. This concerns the coupling of object oriented and real-time features through inheritance.

Recall from the paper on the object oriented model that we can declare an interface thus

```
type Int is interface;
```

An interface is essentially an abstract tagged type that cannot have any components but can have abstract operations and null procedures. We can then derive other interfaces and tagged types by inheritance such as

```
type Another_Int is interface and Int1 and Int2;

type T is new Int1 and Int2;

type TT is new T and Int3 and Int4;
```

Remember that a tagged type can be derived from at most one other normal tagged type but can also be derived from several interfaces. In the list, the first is called the parent (it can be a normal tagged type or an interface) and any others (which can only be interfaces) are called progenitors.

Ada 2005 also introduces further categories of interfaces, namely synchronized, protected, and task interfaces. A synchronized interface can be implemented by either a task or protected type; a protected interface can only be implemented by a protected type and a task interface can only be implemented by a task type.

A nonlimited interface can only be implemented by a nonlimited type. However, an explicitly marked limited interface can be implemented by any tagged type (limited or not) or by a protected or task type. Remember that task and protected types are inherently limited. Note that we use the term limited interface to refer collectively to interfaces marked limited, synchronized, task or protected and we use explicitly limited to refer to those actually marked as limited.

So we can write

```
type LI is limited interface;                    -- similarly type LI2

type SI is synchronized interface;

type TI is task interface;

type PI is protected interface;
```

and we can of course provide operations which must be abstract or null. (Remember that **synchronized** is a new reserved word.)

We can compose these interfaces provided that no conflict arises. The following are all permitted:

```
type TI2 is task interface and LI and TI;

type LI3 is limited interface and LI and LI2;

type TI3 is task interface and LI and LI2;

type SI2 is synchronized interface and LI and SI;
```

The rule is simply that we can compose two or more interfaces provided that we do not mix task and protected interfaces and the resulting interface must be not earlier in the hierarchy: limited, synchronized, task/protected than any of the ancestor interfaces.

We can derive a real task type or protected type from one or more of the appropriate interfaces

```
task type TT is new TI with
  ...                          -- and here we give entries as usual
end TT;
```

or

```
protected type PT is new LI and SI with
  ...
end PT;
```

Unlike tagged record types we cannot derive a task or protected type from another task or protected type as well. So the derivation hierarchy can only be one level deep once we declare an actual task or protected type.

The operations of these various interfaces are declared in the usual way and an interface composed of several interfaces has the operations of all of them with the same rules regarding duplication and overriding of an abstract operation by a null one and so on as for normal tagged types.

When we declare an actual task or protected type then we must implement all of the operations of the interfaces concerned. This can be done in two ways, either by declaring an entry or protected operation in the specification of the task or protected object or by declaring a distinct subprogram in the same list of declarations (but not both). Of course, if an operation is null then it can be inherited or overridden as usual.

Thus the interface

```
package Pkg is
  type TI is task interface;
  procedure P(X: in TI) is abstract;
  procedure Q(X: in TI; I: in Integer) is null;
end Pkg;
```

could be implemented by

```
package PT1 is
  task type TT1 is new TI with
```

```
      entry P;                          -- P and Q implemented by entries
      entry Q(I: in Integer);
    end TT1;
  end PT1;
```

or by

```
  package PT2 is
    task type TT2 is new TI with
      entry P;                          -- P implemented by an entry
    end TT2;
                                        -- Q implemented by a procedure
    procedure Q(X: in TT2; I: in Integer);
  end PT2;
```

or even by

```
  package PT3 is
    task type TT3 is new TI with end;
                                        -- P implemented by a procedure
                                        -- Q inherited as a null procedure
    procedure P(X: in TT3);
  end PT3;
```

In this last case there are no entries and so we have the juxtaposition **with end** which is somewhat similar to the juxtaposition **is end** that occurs with generic packages used as signatures.

Observe how the first parameter which denotes the task is omitted if it is implemented by an entry. This echoes the new prefixed notation for calling operations of tagged types in general. Remember that rather than writing

```
Op(X, Y, Z, ...);
```

we can write

```
X.Op(Y, Z, ...);
```

provided certain conditions hold such as that X is of a tagged type and that Op is a primitive operation of that type.

In order for the implementation of an interface operation by an entry of a task type or a protected operation of a protected type to be possible some fairly obvious conditions must be satisfied.

In all cases the first parameter of the interface operation must be of the task type or protected type (it may be an access parameter).

In addition, in the case of a protected type, the first parameter of an operation implemented by a protected procedure or entry must have mode **out** or **in out** (and in the case of an access parameter it must be an access to variable parameter).

If the operation does not fit these rules then it has to be implemented as a subprogram. An important example is that a function has to be implemented as a function in the case of a task type because there is no such thing as a function entry. However, a function can often be directly implemented as a protected function in the case of a protected type.

Entries and protected operations which implement inherited operations may be in the visible part or private part of the task or protected type in the same way as for tagged record types.

It may seem rather odd that an operation can be implemented by a subprogram that is not part of the task or protected type itself – it seems as if it might not be task safe in some way. But a common

paradigm is where an operation as an abstraction has to be implemented by two or more entry calls. An example occurs in some implementations of the classic readers and writers problem as we shall see later.

Of course a task or protected type which implements an interface can have additional entries and operations as well just as a derived tagged type can have more operations than its parent.

The overriding indicators **overriding** and **not overriding** can be applied to entries as well as to procedures. Thus the package PT2 above could be written as

```
package PT2 is
  task type TT2 is new TI with
    overriding                    -- P implemented by an entry
    entry P;
  end TT2;

  overriding                      -- Q implemented by procedure
  procedure Q(X: in TT2; I: in Integer);
end PT2;
```

We will now explore a simple readers and writers example in order to illustrate various points. We start with the following interface

```
package RWP is
  type RW is limited interface;
  procedure Write(Obj: out RW; X: in Item) is abstract;
  procedure Read(Obj: in RW; X: out Item) is abstract;
end RWP;
```

The intention here is that the interface describes the abstraction of providing an encapsulation of a hidden location and a means of writing a value (of some type Item) to it and reading a value from it – very trivial.

We could implement this in a nonsynchronized manner thus

```
type Simple_RW is new RW with
  record
    V: Item;
  end record;

overriding
procedure Write(Obj: out Simple_RW; X: in Item);

overriding
procedure Read(Obj: in Simple_RW; X: out Item);

...

procedure Write(Obj: out Simple_RW; X: in Item) is
begin
  Obj.V := X;
end Write;

procedure Read(Obj: in Simple_RW; X: out Item) is
begin
  X := Obj.V;
end Read;
```

This implementation is of course not task safe (task safe is sometimes referred to as thread-safe). If a task calls Write and the type Item is a composite type and the writing task is interrupted part of the way through writing, then a task which calls Read might get a curious result consisting of part of the new value and part of the old value.

For illustration we could derive a synchronized interface

```
type Sync_RW is synchronized interface and RW;
```

This interface can only be implemented by a task or protected type. For a protected type we might have

```
protected type Prot_RW is new Sync_RW with
  overriding
  procedure Write(X: in Item);
  overriding
  procedure Read(X: out Item);
private
  V: Item;
end;

protected body Prot_RW is
  procedure Write(X: in Item) is
  begin
    V := X;
  end Write;

  procedure Read(X: out Item) is
  begin
    X := V;
  end Read;
end Prot_RW;
```

Again observe how the first parameter of the interface operations is omitted when they are implemented by protected operations.

This implementation is perfectly task safe. However, one of the characteristics of the readers and writers example is that it is quite safe to allow multiple readers since they cannot interfere with each other. But the type Prot_RW does not allow multiple readers because protected procedures can only be executed by one task at a time.

Now consider

```
protected type Multi_Prot_RW is new Sync_RW with
  overriding
  procedure Write(X: in Item);
  not overriding
  function Read return Item;
private
  V: Item;
end;

overriding
procedure Read(Obj: in Multi_Prot_RW; X: out Item);

...
```

```
protected body Multi_Prot_RW is
  procedure Write(X: in Item) is
  begin
    V := X;
  end Write;

  function Read return Item is
  begin
    return V;
  end Read;
end Multi_Prot_RW;

procedure Read(Obj: in Multi_Prot_RW; X: out Item) is
begin
  X := Obj.Read;
end Read;
```

In this implementation the procedure Read is implemented by a procedure outside the protected type and this procedure then calls the function Read within the protected type. This allows multiple readers because one of the characteristics of protected functions is that multiple execution is permitted (but of course calls of the protected procedure Write are locked out while any calls of the protected function are in progress). The structure is emphasized by the use of overriding indicators.

A simple tasking implementation might be as follows

```
task type Task_RW is new Sync_RW with
  overriding
  entry Write(X: in Item);
  overriding
  entry Read(X: out Item);
end;

task body Task_RW is
  V: Item;
begin
  loop
    select
      accept Write(X: in Item) do
        V := X;
      end Write;
    or
      accept Read(X: out Item) do
        X := V;
      end Read;
    or
      terminate;
    end select;
  end loop;
end Task_RW;
```

Finally, here is a tasking implementation which allows multiple readers and ensures that an initial value is set by only allowing a call of Write first. It is based on an example in that textbook [3].

```
task type Multi_Task_RW(V: access Item) is new Sync_RW with
  overriding
  entry Write(X: in Item);
```

```
   not overriding
   entry Start;
   not overriding
   entry Stop;
end;

overriding
procedure Read(Obj: in Multi_Task_RW; X: out Item);

...

task body Multi_Task_RW is
   Readers: Integer := 0;
begin
   accept Write(X: in Item) do
      V.all := X;
   end Write;
   loop
      select
         when Write'Count = 0 =>
         accept Start;
         Readers := Readers + 1;
      or
         accept Stop;
         Readers := Readers – 1;

      or
         when Readers = 0 =>
         accept Write(X: in Item) do
            V.all := X;
         end Write;
      or
         terminate;
      end select;
   end loop;
end Multi_Task_RW;

overriding
procedure Read(Obj: in Multi_Task_RW; X: out Item) is
begin
   Obj.Start;
   X := Obj.V.all;
   Obj.Stop;
end Read;
```

In this case the data being protected is accessed via the access discriminant of the task. It is structured this way so that the procedure Read can read the data directly. Note also that the procedure Read (which is the implementation of the procedure Read of the interface) calls two entries of the task.

It should be observed that this last example is by way of illustration only. As is well known, the Count attribute used in tasks (as opposed to protected objects) can be misleading if tasks are aborted or if entry calls are timed out. Moreover, it would be gruesomely slow.

So we have seen that a limited interface such as RW might be implemented by a normal tagged type (plus its various operations) and by a protected type and also by a task type. We could then dispatch

to the operations of any of these according to the tag of the type concerned. Observe that task and protected types are now other forms of tagged types and so we have to be careful to say tagged record type (or informally, normal tagged type) where appropriate.

In the above example, the types Simple_RW, Prot_RW, Multi_Prot_RW, Task_RW and Multi_Task_RW all implement the interface RW.

So we might have

```
RW_Ptr: access RW'Class := ...

...
RW_Ptr.Write(An_Item);                -- dispatches
```

and according to the value in RW_Ptr this might call the appropriate entry or procedure of an object of any of the types implementing the interface RW.

However if we have

```
Sync_RW_Ptr: access Sync_RW'Class := ...
```

then we know that any implementation of the synchronized interface Sync_RW will be task safe because it can only be implemented by a task or protected type. So the dispatching call

```
Sync_RW_Ptr.Write(An_Item);           -- task safe dispatching
```

will be task safe.

An interesting point is that because a dispatching call might be to an entry or to a procedure we now permit what appear to be procedure calls in timed entry calls if they might dispatch to an entry.

So we could have

```
select
  RW_Ptr.Read(An_Item);               -- dispatches
or
  delay Seconds(10);
end select;
```

Of course it might dispatch to the procedure Read if the type concerned turns out to be Simple_RW in which case a time out could not occur. But if it dispatched to the entry Read of the type Task_RW then it could time out.

On the other hand we are not allowed to use a timed call if it is statically known to be a procedure. So

```
A_Simple_Object: Simple_RW;
...
select
  A_Simple_Object.Read(An_Item);      -- illegal
or
  delay Seconds(10);
end select;
```

is not permitted.

A note of caution is in order. Remember that the time out is to when the call gets accepted. If it dispatches to Multi_Task_RW.Read then time out never happens because the Read itself is a procedure and gets called at once. However, behind the scenes it calls two entries and so could take a long time. But if we called the two entries directly with timed calls then we would get a time out if there were a lethargic writer in progress. So the wrapper distorts the abstraction. In a sense this is

not much worse than the problem we have anyway that a time out is to when a call is accepted and not to when it returns – it could hardly be otherwise.

The same rules apply to conditional entry calls and also to asynchronous select statements where the triggering statement can be a dispatching call.

In a similar way we also permit timed calls on entries renamed as procedures. But note that we do not allow timed calls on generic formal subprograms even though they might be implemented as entries.

Another important point to note is that we can as usual assume the common properties of the class concerned. Thus in the case of a task interface we know that it must be implemented by a task and so the operations such as **abort** and the attributes Identity, Callable and so on can be applied. If we know that an interface is synchronized then we do know that it has to be implemented by a task or a protected type and so is task safe.

Typically an interface is implemented by a task or protected type but it can also be implemented by a singleton task or protected object despite the fact that singletons have no type name. Thus we might have

```
protected An_RW is new Sync_RW with
  procedure Write(X: in Item);
  procedure Read(X: out Item);
end;
```

with the obvious body. However we could not declare a single protected object similar to the type Multi_Prot_RW above. This is because we need a type name in order to declare the overriding procedure Read outside the protected object. So singleton implementations are possible provided that the interface can be implemented directly by the task or protected object without external subprograms.

Here is another example

```
type Map is protected interface;
procedure Put(M: Map; K: Key; V: Value) is abstract;
```

can be implemented by

```
protected A_Map is new Map with
  procedure Put(K: Key; V: Value);
  ...
end A_Map;
```

There is a fairly obvious rule about private types and synchronized interfaces. Both partial and full view must be synchronized or not. Thus if we wrote

```
type SI is synchronized interface;
type T is new SI with private;
```

then the full type T has to be a task type or protected type or possibly a synchronized, protected or task interface.

We conclude this discussion on interfaces by saying a few words about the use of the word limited. (Much of this has already been explained in the paper on the object oriented model but it is worth repeating in the context of concurrent types.) We always explicitly insert limited, synchronized, task, or protected in the case of a limited interface in order to avoid confusion. So to derive a new explicitly limited interface from an existing limited interface LI we write

```
type LI2 is limited interface and LI;
```

whereas in the case of normal types we can write

```
type LT is limited ...
type LT2 is new LT and LI with ...          -- LT2 is limited
```

then LT2 is limited by the normal derivation rules. Types take their limitedness from their parent (the first one in the list, provided it is not a progenitor) and it does not have to be given explicitly on type derivation – although it can be in Ada 2005 thus

```
type LT2 is limited new LT and LI with ...
```

Remember the important rule that all descendants of a nonlimited interface have to be nonlimited because otherwise limited types could end up with an assignment operation.

This means that we cannot write

```
type NLI is interface;                      -- nonlimited
type LI is limited interface;               -- limited
task type TT is new NLI and LI with ...     -- illegal
```

This is illegal because the interface NLI in the declaration of the task type TT is not limited.

4 The Ravenscar profile

The purpose of the Ravenscar profile is to restrict the use of many tasking facilities so that the effect of the program is predictable. The profile was defined by the International Real-Time Ada Workshops which met twice at the remote village of Ravenscar on the coast of Yorkshire in North-East England. A general description of the principles and use of the profile in high integrity systems will be found in an ISO/IEC Technical Report [2] and so we shall not cover that material here.

Here is a historical interlude. It is reputed that the hotel in which the workshops were held was originally built as a retreat for King George III to keep a mistress. Another odd rumour is that he ordered all the natural trees to be removed and replaced by metallic ones whose metal leaves clattered in the wind. It also seems that Henry Bolingbroke landed at Ravenscar in July 1399 on his way to take the throne as Henry IV. Ravenscar is mentioned several times by Shakespeare in Act II of King Richard II; it is spelt Ravenspurg which is slightly confusing – maybe we need the ability to rename profile identifiers.

A profile is a mode of operation and is specified by the pragma Profile which defines the particular profile to be used. The syntax is

```
pragma Profile(profile_identifier [ , profile_argument_associations]);
```

where profile_argument_associations is simply a list of pragma argument associations separated by commas.

Thus to ensure that a program conforms to the Ravenscar profile we write

```
pragma Profile(Ravenscar);
```

The general idea is that a profile is equivalent to a set of configuration pragmas.

In the case of Ravenscar the pragma is equivalent to the joint effect of the following pragmas

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;

pragma Restrictions(
   No_Abort_Statements,
```

```
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Local_Timing_Events,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Specific_Termination_Handlers,
No_Task_Allocators,
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers,
Max_Entry_Queue_Length => 1,
Max_Protected_Entries => 1,
Max_Task_Entries => 0,
No_Dependence => Ada.Asynchronous_Task_Control,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Execution_Time.Group_Budget,
No_Dependence => Ada.Execution_Time.Timers,
No_Dependence => Ada.Task_Attributes);
```

The pragma Detect_Blocking plus many of the Restrictions identifiers are new to Ada 2005. These will now be described.

The pragma Detect_Blocking, as its name implies, ensures that the implementation will detect a potentially blocking operation in a protected operation and raise Program_Error. Without this pragma the implementation is not required to detect blocking and so tasks might be locked out for an unbounded time and the program might even deadlock.

The identifier No_Dynamic_Attachment means that there are no calls of the operations in the package Ada.Interrupts.

The identifier No_Dynamic_Priorities means that there is no dependence on the package Ada.Priorities as well as no uses of the attribute Priority (this is a new attribute for protected objects as explained at the end of this section).

Note that the rules are that you cannot read as well as not write the priorities – this applies to both the procedure for reading task priorities and reading the attribute for protected objects.

The identifier No_Local_Protected_Objects means that protected objects can only be declared at library level and the identifier No_Protected_Type_Allocators means that there are no allocators for protected objects or objects containing components of protected types.

The identifier No_Local_Timing_Events means that objects of the type Timing_Event in the package Ada.Real_Time.Timing_Events can only be declared at library level. This package is described in Section 6 below.

The identifiers No_Relative_Delay, No_Requeue_Statements, and No_Select_Statements mean that there are no relative delay, requeue or select statements respectively.

The identifier No_Specific_Termination_Handlers means that there are no calls of the procedure Set_Specific_Handler or the function Specific_Handler in the package Task_Termination and the identifier No_Task_Termination means that all tasks should run for ever. Note that we are permitted to set a fallback handler so that if any task does attempt to terminate then it will be detected.

The identifier Simple_Barriers means that the Boolean expression in a barrier of an entry of a protected object shall be either a static expression (such as True) or a Boolean component of the protected object itself.

The Restrictions identifier Max_Entry_Queue_Length sets a limit on the number of calls permitted on an entry queue. It is an important property of the Ravenscar profile that only one call is permitted at a time on an entry queue of a protected object.

The identifier No_Dependence is not specific to the Real-Time Systems annex and is properly described in the next paper. In essence it indicates that the program does not depend upon the given language defined package. In this case it means that a program conforming to the Ravenscar profile cannot use any of the packages Asynchronous_Task_Control, Calendar, Execution_Time.Group_Budget, Execution_Time.Timers and Task_Attributes. Some of these packages are new and are described later in this paper.

Note that No_Dependence cannot be used for No_Dynamic_Attachment because that would prevent use of the child package Ada.Interrupts.Names.

All the other restrictions identifiers used by the Ravenscar profile were already defined in Ada 95. Note also that the identifier No_Asynchronous_Control has been moved to Annex J because it can now be replaced by the use of No_Dependence.

5 Scheduling and dispatching

Another area of increased flexibility in Ada 2005 is that of task dispatching policies. In Ada 95, the only predefined policy is FIFO_Within_Priorities although other policies are permitted. Ada 2005 provides further pragmas, policies and packages which facilitate many different mechanisms such as non-preemption within priorities, the familiar Round Robin using timeslicing, and the more recently acclaimed Earliest Deadline First (EDF) policy. Moreover it is possible to mix different policies according to priority level within a partition.

In order to accommodate these many changes, Section D.2 (Priority Scheduling) of the Reference Manual has been reorganized as follows

D.2.1 The Task Dispatching Model
D.2.2 Task Dispatching Pragmas
D.2.3 Preemptive Dispatching
D.2.4 Non-Preemptive Dispatching
D.2.5 Round Robin Dispatching
D.2.6 Earliest Deadline First Dispatching

Overall control is provided by two pragmas. They are

pragma Task_Dispatching_Policy(*policy*_identifier);

pragma Priority_Specific_Dispatching(*policy*_identifer,
*first_priority*_expression, *last_priority*_expression);

The pragma Task_Dispatching_Policy, which already exists in Ada 95, applies the same policy throughout a whole partition. The pragma Priority_Specific_Dispatching, which is new in Ada 2005, can be used to set different policies for different ranges of priority levels.

The full set of predefined policies in Ada 2005 is

FIFO_Within_Priorities – This already exists in Ada 95. Within each priority level to which it applies tasks are dealt with on a first-in-first-out basis. Moreover, a task may preempt a task of a lower priority.

Non_Preemptive_FIFO_Within_Priorities – This is new in Ada 2005. Within each priority level to which it applies tasks run to completion or until they are blocked or execute a delay statement. A task cannot be preempted by one of higher priority. This sort of policy is widely used in high integrity applications.

Round_Robin_Within_Priorities – This is new in Ada 2005. Within each priority level to which it applies tasks are timesliced with an interval that can be specified. This is a very traditional policy widely used since the earliest days of concurrent programming.

EDF_Across_Priorities – This is new in Ada 2005. This provides Earliest Deadline First dispatching. The general idea is that within a range of priority levels, each task has a deadline and that with the earliest deadline is processed. This is a fashionable new policy and has mathematically provable advantages with respect to efficiency.

For further details of these policies consult the forthcoming book by Alan Burns and Andy Wellings [4].

These various policies are controlled by the package Ada.Dispatching plus two child packages. The root package has specification

```
package Ada.Dispatching is
   pragma Pure(Dispatching);
   Dispatching_Policy_Error: exception;
end Ada.Dispatching;
```

As can be seen this root package simply declares the exception Dispatching_Policy_Error which is used by the child packages.

The child package Round_Robin enables the setting of the time quanta for time slicing within one or more priority levels. Its specification is

```
with System;  use System;
with Ada.Real_Time;  use Ada.Real_Time;
package Ada.Dispatching.Round_Robin is
   Default_Quantum: constant Time_Span := implementation-defined;
   procedure Set_Quantum(Pri: in Priority, Quantum: in Time_Span);
   procedure Set_Quantum(Low, High: in Priority; Quantum: in Time_Span);
   function Actual_Quantum(Pri: Priority) return Time_Span;
   function Is_Round_Robin(Pri: Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

The procedures Set_Quantum enable the time quantum to be used for time slicing to be set for one or a range of priority levels. The default value is of course the constant Default_Quantum. The function Actual_Quantum enables us to find out the current value of the quantum being used for a particular priority level. Its identifier reflects the fact that the implementation may not be able to apply the exact actual value given in a call of Set_Quantum. The function Is_Round_Robin enables us to check whether the round robin policy has been applied to the given priority level. If we attempt to do something stupid such as set the quantum for a priority level to which the round robin policy does not apply then the exception Dispatching_Policy_Error is raised.

The other new policy concerns deadlines and is controlled by a new pragma Relative_Deadline and the child package Dispatching.EDF. The syntax of the pragma is

```
pragma Relative_Deadline(relative_deadline_expression);
```

The deadline of a task is a property similar to priority and both are used for scheduling. Every task has a priority of type Integer and every task has a deadline of type Ada.Real_Time.Time. Priorities can be set when a task is created by pragma Priority

```
task T is
  pragma Priority(P);
```

and deadlines can similarly be set by the pragma Relative_Deadline thus

```
task T is
  pragma Relative_Deadline(RD);
```

The expression RD has type Ada.Real_Time.Time_Span. Note carefully that the pragma sets the relative and not the absolute deadline. The initial absolute deadline of the task is

```
Ada.Real_Time.Clock + RD
```

where the call of Clock is made between task creation and the start of its activation.

Both pragmas Priority and Relative_Deadline can appear in the main subprogram and they then apply to the environment task. If they appear in any other subprogram then they are ignored. Both properties can also be set via a discriminant. In the case of priorities we can write

```
task type TT(P: Priority) is
  pragma Priority(P);
  ...
end;

High_Task: TT(13);
Low_Task: TT(7);
```

We cannot do the direct equivalent for deadlines because Time_Span is private and so not discrete. We have to use an access discriminant thus

```
task type TT(RD: access Timespan) is
  pragma Relative_Deadline(RD.all);
  ...
end;

One_Sec: aliased constant Time_Span := Seconds(1);
Ten_Mins: aliased constant Time_Span := Minutes(10);

Hot_Task: TT(One_Sec'Access);
Cool_Task: TT(Ten_Mins'Access);
```

Note incidentally that functions Seconds and Minutes have been added to the package Ada.Real_Time. Existing functions Nanoseconds, Microseconds and Milliseconds in Ada 95 enable the convenient specification of short real time intervals (values of type Time_Span). However, the specification of longer intervals such as four minutes meant writing something like Milliseconds(240_000) or perhaps 4*60*Milliseconds(1000). In view of the fact that EDF scheduling and timers (see Section 6) would be likely to require longer times the functions Seconds and Minutes are added in Ada 2005. There is no function Hours because the range of time spans is only guaranteed to be 3600 seconds anyway.

If a task is created and it does not have a pragma Priority then its initial priority is that of the task that created it. If a task does not have a pragma Relative_Deadline then its initial absolute deadline is the constant Default_Deadline in the package Ada.Dispatching.EDF; this constant has the value Ada.Real_Time.Time_Last (effectively the end of the universe).

Priorities can be dynamically manipulated by the subprograms in the package Ada.Dynamic_Priorities and deadlines can similarly be manipulated by the subprograms in the package Ada.Dispatching.EDF whose specification is

```
with Ada.Real_Time;  use Ada.Real_Time;
with Ada.Task_Identification; use Ada.Task_Identification;
package Ada.Dispatching.EDF is
   subtype Deadline is Ada.Real_Time.Time;
   Default_Deadline: constant Deadline := Time_Last;
   procedure Set_Deadline(D: in Deadline; T: in Task_Id := Current_Task);
   procedure Delay_Until_And_Set_Deadline (Delay_Until_Time: in Time;
                                           Deadline_Offset: in Time_Span);
   function Get_Deadline(T: Task_Id := Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

The subtype Deadline is just declared as a handy abbreviation. The constant Default_Deadline is set to the end of the universe as already mentioned. The procedure Set_Deadline sets the deadline of the task concerned to the value of the parameter D. The long-winded Delay_Until_And_Set_Deadline delays the task concerned until the value of Delay_Until_Time and sets its deadline to be the interval Deadline_Offset from that time – this is useful for periodic tasks. The function Get_Deadline enables us to find the current deadline of a task.

It is important to note that this package can be used to set and retrieve deadlines for tasks whether or not they are subject to EDF dispatching. We could for example use an ATC on a deadline overrun (ACT = Asynchronous Transfer of Control using a select statement). Hence there is no function Is_EDF corresponding to Is_Round_Robin and calls of the subprograms in this package can never raise the exception Dispatching_Policy_Error.

If we attempt to apply one of the subprograms in this package to a task that has already terminated then Tasking_Error is raised. If the task parameter is Null_Task_Id then Program_Error is raised.

As mentioned earlier, a policy can be selected for a whole partition by for example

```
pragma Task_Dispatching_Policy(Round_Robin_Within_Priorities);
```

whereas in order to mix different policies across different priority levels we can write

```
pragma Priority_Specific_Dispatching(Round_Robin_Within_Priority, 1, 1);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
pragma Priority_Specific_Dispatching(FIFO_Within_Priority, 11, 24);
```

This sets Round Robin at priority level 1, EDF at levels 2 to 10, and FIFO at levels 11 to 24. This means for example that none of the EDF tasks can run if any of the FIFO ones can. In other words if any tasks in the highest group can run then they will do so and none in the other groups can run. The scheduling within a range takes over only if tasks in that range can go and none in the higher ranges can.

Note that if we write

```
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 5);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 6, 10);
```

then this is not the same us

```
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
```

despite the fact that the two ranges in the first case are contiguous. This is because in the first case any task in the 6 to 10 range will take precedence over any task in the 2 to 5 range whatever the deadlines. If there is just one range then only the deadlines count in deciding which tasks are scheduled.

This is emphasized by the fact that the policy name uses Across rather than Within. For other policies such as Round_Robin_Within_Priority two contiguous ranges would be the same as a single range.

We conclude this section with a few words about ceiling priorities.

In Ada 95, the priority of a task can be changed but the ceiling priority of a protected object cannot be changed. It is permanently set when the object is created using the pragma Priority. This is often done using a discriminant so that at least different objects of a given protected type can have different priorities. Thus we might have

```
protected type PT(P: Priority) is
  pragma Priority(P);
  ...
end PT;

PO: PT(7);                          -- ceiling priority is 7
```

The fact that the ceiling priority of a protected object is static can be a nuisance in many applications especially when the priority of tasks can be dynamic. A common workaround is to give a protected object a higher ceiling than needed in all circumstances (often called "the ceiling of ceilings"). This results in tasks having a higher active priority than necessary when accessing the protected object and this can interfere with the processing of other tasks in the system and thus upset overall schedulability. Moreover, it means that a task of high priority can access an object when it should not (if a task with a priority higher than the ceiling priority of a protected object attempts to access the object then Program_Error is raised – if the object has an inflated priority then this check will pass when it should not).

This difficulty is overcome in Ada 2005 by allowing protected objects to change their priority. This is done through the introduction of an attribute Priority which applies just to protected objects. It can only be accessed within the body of the protected object concerned.

As an example a protected object might have a procedure to change its ceiling priority by a given amount. This could be written as follows

```
protected type PT is
  procedure Change_Priority(Change: in Integer);
  ...
end;

protected body PT is
  procedure Change_Priority(Change: in Integer) is
  begin
    ...                             -- PT'Priority has old value here
    PT'Priority := PT'Priority + Change;
    ...                             -- PT'Priority has new value here
    ...
  end Change_Priority;
  ...
end PT;
```

Changing the ceiling priority is thus done while mutual exclusion is in force. Although the value of the attribute itself is changed immediately the assignment is made, the actual ceiling priority of the protected object is only changed when the protected operation (in this case the call of Change_ Priority) is finished.

Note the unusual syntax. Here we permit an attribute as the destination of an assignment statement. This happens nowhere else in the language. Other forms of syntax were considered but this seemed the most expressive.

6 CPU clocks and timers

Ada 2005 introduces three different kinds of timers. Two are concerned with monitoring the CPU time of tasks – one applies to a single task and the other to groups of tasks. The third timer measures real time rather than execution time and can be used to trigger events at specific real times. We will look first at the CPU timers because that introduces more new concepts.

The execution time of one or more tasks can be monitored and controlled by the new package Ada.Execution_Time plus two child packages.

Ada.Execution_Time – this is the root package and enables the monitoring of execution time of individual tasks.

Ada.Execution_Time.Timers – this provides facilities for defining and enabling timers and for establishing a handler which is called by the run time system when the execution time of the task reaches a given value.

Ada.Execution_Time.Group_Budgets – this enables several tasks to share a budget and provides means whereby action can be taken when the budget expires.

The execution time of a task, or CPU time as it is commonly called, is the time spent by the system executing the task and services on its behalf. CPU times are represented by the private type CPU_Time. This type and various subprograms are declared in the root package Ada.Execution_Time whose specification is as follows (as before we have added some use clauses in order to ease the presentation)

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

   type CPU_Time is private;
   CPU_Time_First: constant CPU_Time;
   CPU_Time_Last: constant CPU_Time;
   CPU_Time_Unit: constant := implementation-defined-real-number;
   CPU_Tick: constant Time_Span;

   function Clock(T: Task_Id := Current_Task) return CPU_Time;

   function "+" (Left: CPU_Time; Right: Time_Span) return CPU_Time;
   function "+" (Left: Time_Span; Right: CPU_Time) return CPU_Time;
   function "–" (Left: CPU_Time; Right: Time_Span) return CPU_Time;
   function "–" (Left: CPU_Time; Right: CPU_Time) return Time_Span;

   function "<" (Left, Right: CPU_Time) return Boolean;
   function "<=" (Left, Right: CPU_Time) return Boolean;
   function ">" (Left, Right: CPU_Time) return Boolean;
   function ">=" (Left, Right: CPU_Time) return Boolean;

   procedure Split(T: in CPU_Time; SC: out Seconds_Count; TS: out Time_Span);
   function Time_Of(SC: Seconds_Count; TS: Time_Span := Time_Span_Zero)
                                                          return CPU_Time;
```

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

The CPU time of a particular task is obtained by calling the function Clock with the task as parameter. It is set to zero at task creation.

The constants CPU_Time_First and CPU_Time_Last give the range of values of CPU_Time. CPU_Tick gives the average interval during which successive calls of Clock give the same value and thus is a measure of the accuracy whereas CPU_Time_Unit gives the unit of time measured in seconds. We are assured that CPU_Tick is no greater than one millisecond and that the range of values of CPU_Time is at least 50 years (provided always of course that the implementation can cope).

The various subprograms perform obvious operations on the type CPU_Time and the type Time_Span of the package Ada.Real_Time.

A value of type CPU_Time can be converted to a Seconds_Count plus residual Time_Span by the function Split which is similar to that in the package Ada.Real_Time. The function Time_Of similarly works in the opposite direction. Note the default value of Time_Span_Zero for the second parameter – this enables times of exact numbers of seconds to be given more conveniently thus

```
Four_Secs: CPU_Time := Time_Of(4);
```

In order to find out when a task reaches a particular CPU time we can use the facilities of the child package Ada.Execution_Time.Timers whose specification is

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

   type Timer(T: not null access constant Task_Id) is tagged limited private;
   type Timer_Handler is access protected procedure (TM: in out Timer);

   Min_Handler_Ceiling: constant Any_Priority := implementation-defined;

   procedure Set_Handler(TM: in out Timer; In_Time: Time_Span; Handler: Timer_Handler);
   procedure Set_Handler(TM: in out Timer; At_Time: CPU_Time; Handler: Timer_Handler);

   function Current_Handler(TM: Timer) return Timer_Handler;
   procedure Cancel_Handler(TM: in out Timer; Cancelled: out Boolean);
   function Time_Remaining(TM: Timer) return Time_Span;

   Timer_Resource_Error: exception;

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

The general idea is that we declare an object of type Timer whose discriminant identifies the task to be monitored – note the use of **not null** and **constant** in the discriminant. We also declare a protected procedure which takes the timer as its parameter and which performs the actions required when the CPU_Time of the task reaches some value. Thus to take some action (perhaps abort for example although that would be ruthless) when the CPU_Time of the task My_Task reaches 2.5 seconds we might first declare

```
My_Timer: Timer(My_Task'Identity'Access);
Time_Max: CPU_Time := Time_Of(2, Milliseconds(500));
```

and then

```
protected Control is
  procedure Alarm(TM: in out Timer);
end;

protected body Control is
  procedure Alarm(TM: in out Timer) is
  begin
    -- abort the task
    Abort_Task(TM.T.all);
  end Alarm;
end Control;
```

Finally we set the timer in motion by calling the procedure Set_Handler which takes the timer, the time value and (an access to) the protected procedure thus

```
Set_Handler(My_Timer, Time_Max, Control.Alarm'Access);
```

and then when the CPU time of the task reaches Time_Max, the protected procedure Control.Alarm is executed. Note how the timer object incorporates the information regarding the task concerned using an access discriminant T and that this is passed to the handler via its parameter TM.

Aborting the task is perhaps a little violent. Another possibility is simply to reduce its priority so that it is no longer troublesome, thus

```
    -- cool that task
    Set_Priority(Priority'First, TM.T.all);
```

Another version of Set_Handler enables the timer to be set for a given interval (of type Time_Span).

The handler associated with a timer can be found by calling the function Current_Handler. This returns null if the timer is not set in which case we say that the timer is clear.

When the timer expires, and just before calling the protected procedure, the timer is set to the clear state. One possible action of the handler, having perhaps made a note of the expiration of the timer, it to set the handler again or perhaps another handler. So we might have

```
protected body Control is
  procedure Alarm(TM: in out Timer) is
  begin
    Log_Overflow(TM);                -- note that timer had expired
    -- and then reset it for another 500 milliseconds
    Set_Handler(TM, Milliseconds(500), Kill'Access);
  end Alarm;

  procedure Kill(TM: in out Timer) is
  begin
    -- expired again so kill it
    Abort_Task(TM.T.all);
  end Kill;
end Control;
```

In this scenario we make a note of the fact that the task has overrun and then give it another 500 milliseconds but with the handler Control.Kill so that the second time is the last chance.

Setting the value of 500 milliseconds directly in the call is a bit crude. It might be better to parameterize the protected type thus

```
protected type Control(MS: Integer) is ...
...
My_Control: Control(500);
```

and then the call of Set_Handler in the protected procedure Alarm would be

```
Set_Handler(TM, Milliseconds(MS), Kill'Access);
```

Observe that overload resolution neatly distinguishes whether we are calling Set_Handler with an absolute time or a relative time.

The procedure Cancel_Handler can be used to clear a timer. The out parameter Cancelled is set to True if the timer was in fact set and False if it was clear. The function Time_Remaining returns Time_Span_Zero if the timer is not set and otherwise the time remaining.

Note also the constant Min_Handler_Ceiling. This is the minimum ceiling priority that the protected procedure should have to ensure that ceiling violation cannot occur.

This timer facility might be implemented on top of a POSIX system. There might be a limit on the number of timers that can be supported and an attempt to exceed this limit will raise Timer_Resource_Error.

We conclude by summarizing the general principles. A timer can be set or clear. If it is set then it has an associated (non-null) handler which will be called after the appropriate time. The key subprograms are Set_Handler, Cancel_Handler and Current_Handler. The protected procedure has a parameter which identifies the event for which it has been called. The same protected procedure can be the handler for many events. The same general structure applies to other kinds of timers which will now be described.

In order to program various so-called aperiodic servers it is necessary for tasks to share a CPU budget.

This can be done using the child package Ada.Execution_Time.Group_Budgets whose specification is

```
with System;  use System;
package Ada.Execution_Time.Group_Budgets is

  type Group_Budget is tagged limited private;
  type Group_Budget_Handler is access protected procedure (GB: in out Group_Budget);

  type Task_Array is array (Positive range <>) of Task_Id;

  Min_Handler_Ceiling: constant Any_Priority := implementation-defined;

  procedure Add_Task(GB: in out Group_Budget; T: in Task_Id);
  procedure Remove_Task(GB: in out Group_Budget; T: in Task_Id);
  function Is_Member(GB: Group_Budget; T: Task_Id) return Boolean;
  function Is_A_Group_Member(T: Task_Id) return Boolean;
  function Members(GB: Group_Budget) return Task_Array;

  procedure Replenish(GB: in out Group_Budget; To: in Time_Span);
  procedure Add(GB: in out Group_Budget; Interval: in Time_Span);

  function Budget_Has_Expired(GB: Group_Budget) return Boolean;
  function Budget_Remaining(GB: Group_Budget) return Time_Span;

  procedure Set_Handler(GB: in out Group_Budget; Handler: in Group_Budget_Handler);
  function Current_Handler(GB: Group_Budget) return Group_Budget_Handler;
  procedure Cancel_Handler(GB: in out Group_Budget; Cancelled: out Boolean);
```

```
   Group_Budget_Error: exception;

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

This has much in common with its sibling package Timers but there are a number of important differences.

The first difference is that we are here considering a CPU budget shared among several tasks. The type Group_Budget both identifies the group of tasks it covers and the size of the budget.

Various subprograms enable tasks in a group to be manipulated. The procedures Add_Task and Remove_Task add or remove a task. The function Is_Member identifies whether a task belongs to a specific group whereas Is_A_Group_Member identifies whether a task belongs to any group. A task cannot be a member of more than one group. An attempt to add a task to more than one group or remove it from the wrong group and so on raises Group_Budget_Error. Finally the function Members returns all the members of a group as an array.

The value of the budget (initially Time_Span_Zero) can be loaded by the procedure Replenish and increased by the procedure Add. Whenever a budget is non-zero it is counted down as the tasks in the group execute and so consume CPU time. Whenever a budget goes to Time_Span_Zero it is said to have become exhausted and is not reduced further. Note that Add with a negative argument can reduce a budget – it can even cause it to become exhausted but not make it negative.

The function Budget_Remaining simply returns the amount left and Budget_Has_Expired returns True if the budget is exhausted and so has value Time_Span_Zero.

Whenever a budget *becomes* exhausted (that is when the value transitions to zero) a hander is called if one has been set. A handler is a protected procedure as before and procedures Set_Handler, Cancel_Handler, and function Current_Handler are much as expected. But a major difference is that Set_Handler does not set the time value of the budget since that is done by Replenish and Add. The setting of the budget and the setting of the handler are decoupled in this package. Indeed a handler can be set even though the budget is exhausted and the budget can be counting down even though no handler is set. The reason for the different approach simply reflects the usage paradigm for the feature.

So we could set up a mechanism to monitor the CPU time usage of a group of three tasks TA, TB, and TC by first declaring an object of type Group_Budget, adding the three tasks to the group and then setting an appropriate handler. Finally we call Replenish which sets the counting mechanism going. So we might write

```
ABC: Group_Budget;
...
Add_Task(ABC, TA'Identity);
Add_Task(ABC, TB'Identity);
Add_Task(ABC, TC'Identity);

Set_Handler(ABC, Control.Monitor'Access);
Replenish(ABC, Seconds(10));
```

Remember that functions Seconds and Minutes have been added to the package Ada.Real_Time.

The protected procedure might be

```
protected body Control is
   procedure Monitor(GB: in out Group_Budget) is
   begin
```

```
      Log_Budget;
      Add(GB, Seconds(10));                    -- add more time
    end Monitor;
  end Control;
```

The procedure Monitor logs the fact that the budget was exhausted and then adds a further 10 seconds to it. Remember that the handler remains set all the time in the case of group budgets whereas in the case of the single task timers it automatically becomes cleared and has to be set again if required.

If a task terminates then it is removed from the group as part of the finalization process.

Note that again there is the constant Min_Handler_Ceiling.

The final kind of timer concerns real time rather than CPU time and so is provided by a child package of Ada.Real_Time whereas the timers we have seen so far were provided by child packages of Ada.Execution_Time. The specification of the package Ada.Real_Time.Timing_Events is

```
package Ada.Real_Time.Timing_Events is

  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is access protected procedure (Event: in out Timing_Event);

  procedure Set_Handler(Event: in out Timing_Event; At_Time: Time;
                                    Handler: Timing_Event_Handler);
  procedure Set_Handler(Event: in out Timing_Event; In_Time: Time_Span;
                                    Handler: Timing_Event_Handler);

  function Is_Handler_Set(Event: Timing_Event) return Boolean;
  function Current_Handler(Event: Timing_Event) return Timing_Event_Handler;
  procedure Cancel_Handler(Event: in out Timing_Event; Cancelled: out Boolean);

  function Time_Of_Event(Event: Timing_Event) return Time;

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

This package provides a very low level facility and does not involve Ada tasks at all. It has a very similar pattern to the package Execution_Time.Timers. A handler can be set by Set_Handler and again there are two versions one for a relative time and one for absolute time. There are also subprograms Current_Handler and Cancel_Handler. If no handler is set then Current_Handler returns null.

Set_Handler also specifies the protected procedure to be called when the time is reached. Times are of course specified using the type Real_Time rather than CPU_Time.

A minor difference is that this package has a function Time_Of_Event rather than Time_Remaining.

A simple example was given in the introductory paper. We repeat it here for convenience. The idea is that we wish to ring a pinger when our egg is boiled after four minutes. The protected procedure might be

```
protected body Egg is
  procedure Is_Done(Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;
end Egg;
```

and then

```
Egg_Done: Timing_Event;
Four_Min: Time_Span := Minutes(4);
...
Put_Egg_In_Water;
Set_Handler(Event => Egg_Done, In_Time => Four_Min, Handler => Egg.Is_Done'Access);
-- now read newspaper whilst waiting for egg
```

This is unreliable because if we are interrupted between the calls of Put_Egg_In_Water and Set_Handler then the egg will be boiled for too long. We can overcome this by adding a further procedure to the protected object so that it becomes

```
protected Egg is
  procedure Boil(For_Time: in Time_Span);
  procedure Is_Done(Event: in out Timing_Event);
end Egg;

protected body Egg is

  Egg_Done: Timing_Event;

  procedure Boil (For_Time: in Time_Span) is
  begin
    Put_Egg_In_Water;
    Set_Handler(Egg_Done, For_Time, Is_Done'Access);
  end Boil;

  procedure Is_Done (Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;
end Egg;
```

This is much better. The timing mechanism is now completely encapsulated in the protected object and the procedure Is_Done is no longer visible outside. So all we have to do is

```
Egg.Boil(Minutes(4));
-- now read newspaper whilst waiting for egg
```

Of course if the telephone rings as the pinger goes off and before we have a chance to eat the egg then it still gets overdone. One solution is to eat the egg within the protected procedure Is_Done as well. A gentleman would never let a telephone call disturb his breakfast.

One protected procedure could be used to respond to several events. In the case of the CPU timer the discriminant of the parameter identifies the task; in the case of the group and real-time timers, the parameter identifies the event.

If we want to use the same timer for several events then various techniques are possible. Note that the timers are limited so we cannot test for them directly. However, they are tagged and so can be extended. Moreover, we know that they are passed by reference and that the parameters are considered aliased.

Suppose we are boiling six eggs in one of those French breakfast things with a different coloured holder for each egg. We can write

```
type Colour is (Black, Blue, Red, Green, Yellow, Purple);

Eggs_Done: array (Colour) of aliased Timing_Event;
```

We can then set the handler for the egg in the red holder by something like

```
Set_Handler(Eggs_Done(Red), For_Time, Is_Done'Access);
```

and then the protected procedure might be

```
procedure Is_Done(E: in out Timing_Event) is
begin
  for C in Colour loop
    if E'Access = Eggs_Done(C)'Access then
                                -- egg in holder colour C is ready
      ...
      return;
    end if;
  end loop;
                                -- falls out of loop – unknown event!
  raise Not_An_Egg ;
end Is_Done;
```

Although this does work it is more than a little distasteful to compare access values in this way and moreover requires a loop to see which event occurred.

A much better approach is to use type extension and view conversions. First we extend the type Timing_Event to include additional information about the event (in this case the colour) so that we can identify the particular event from within the handler

```
type Egg_Event is new Timing_Event with
  record
    Event_Colour: Colour;
  end record;
```

We then declare an array of these extended events (they need not be aliased)

```
Eggs_Done: array (Colour) of Egg_Event;
```

We can now call Set_Handler for the egg in the red holder

```
Set_Handler(Eggs_Done(Red), For_Time, Is_Done'Access);
```

This is actually a call on the Set_Handler for the type Egg_Event inherited from Timing_Event. But it is the same code anyway.

Remember that values of tagged types are always passed by reference. This means that from within the procedure Is_Done we can recover the underlying type and so discover the information in the extension. This is done by using view conversions.

In fact we have to use two view conversions, first we convert to the class wide type Timing_Event'Class and then to the specific type Egg_Event. And then we can select the component Event_Colour. In fact we can do these operations in one statement thus

```
procedure Is_Done(E: in out Timing_Event) is
  C: constant Colour := Egg_Event(Timing_Event'Class(E)).Event_Colour;
begin
                                -- egg in holder colour C is ready
  ...
end Is_Done;
```

Note that there is a check on the conversion from the class wide type Timing_Event'Class to the specific type Egg_Event to ensure that the object passed as parameter is indeed of the type

Egg_Event (or a further extension of it). If this fails then Tag_Error is raised. In order to avoid this possibility we can use a membership test. For example

```
procedure Is_Done(E: in out Timing_Event) is
  C: Colour;
begin
  if Timing_Event'Class(E) in Egg_Event then
    C := Egg_Event(Timing_Event'Class(E)).Event_Colour;
                              -- egg in holder colour C is ready
    ...
  else
                              -- unknown event – not an egg event!
    raise Not_An_Egg;
  end if;
end Is_Done;
```

The membership test ensures that the event is of the specific type Egg_Event. We could avoid the double conversion to the class wide type by introducing an intermediate variable.

It is important to appreciate that no dispatching is involved in these operations at all – everything is static apart from the membership test.

Of course, it would have been a little more flexible if the various subprograms took a parameter of type Timing_Event'Class but this would have conflicted with the Restrictions identifier No_Dispatch. Note that Ravenscar itself does not impose No_Dispatch but the restriction is in the High-Integrity annex and thus might be imposed on some high-integrity applications which might nevertheless wish to use timers in a simple manner.

A few minor points of difference between the timers are worth summarizing.

The two CPU timers have a constant Min_Handler_Ceiling. This prevents ceiling violation. It is not necessary for the real-time timer because the call of the protected procedure is treated like an interrupt and thus is at interrupt ceiling level.

The group budget timer and the real-time timer do not have an exception corresponding to Timer_Resource_Error for the single task CPU timer. As mentioned above, it is anticipated that the single timer might be implemented on top of a POSIX system in which case there might be a limit to the number of timers especially since each task could be using several timers. In the group case, a task can only be in one group so the number of group timers is necessarily less than the number of tasks and no limit is likely to be exceeded. In the real-time case the events are simply placed on the delay queue and no other resources are required anyway.

It should also be noted that the group timer could be used to monitor the execution time of a single task. However, a task can only be in one group and so only one timer could be applied to a task that way whereas, as just mentioned, the single CPU timer is quite different since a given task could have several timers set for it to expire at different times. Thus both kinds of timers have their own distinct usage patterns.

7 High Integrity Systems annex

There are a few changes to this annex. The most noticeable is that its title has been changed from Safety and Security to High Integrity Systems. This reflects common practice in that high-integrity is now the accepted general term for systems such as safety-critical systems and security-critical systems.

There are some small changes to reflect the introduction of the Ravenscar profile. It is clarified that tasking is permitted in a high-integrity system provided that it is well controlled through, for example, the use of the Ravenscar profile.

A new pragma Partition_Elaboration_Policy is introduced. Its syntax is

```
pragma Partition_Elaboration_Policy(policy_identifier);
```

Two policy identifiers are predefined, namely, Concurrent and Sequential. The pragma is a configuration pragma and so applies throughout a partition. The default policy is Concurrent.

The normal behaviour in Ada when a program starts is that a task declared at library level is activated by the environment task and can begin to execute before all library level elaboration is completed and before the main subprogram is called by the environment task. Race conditions can arise especially when several library tasks are involved. Problems also arise with the attachment of interrupt handlers.

If the policy Sequential is specified then the rules are changed. The following things happen in sequence

- The elaboration of all library units takes place (this is done by the environment task) but library tasks are not activated (we say their activation is deferred). Similarly the attachment of interrupt handlers is deferred.
- The environment task then attaches the interrupts.
- The library tasks are then activated. While this is happening the environment task is suspended.
- Finally, the environment task then executes the main subprogram in parallel with the executing tasks.

Note that from the library tasks' point of view they go seamlessly from activation to execution. Moreover, they are assured that all library units will have been elaborated and all handlers attached before they execute.

If Sequential is specified then

```
pragma Restrictions(No_Task_Hierarchy);
```

must also be specified. This ensures that all tasks are at library level.

A final small point is that the Restrictions identifiers No_Unchecked_Conversion and No_Unchecked_Deallocation are now banished to Annex J because No_Dependence can be used instead.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] ISO/IEC TR 24718:2004 (2004) *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. This is based on University of York Technical Report YCS-2003-348 (2003).

[3] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

[4] A. Burns and A. Wellings (2006) *Concurrent and Real-Time Programming In Ada 2005*, Cambridge University Press.