AI-00297 Timing Events

!standard D.7 (10) 05-03-23 AI95-00297/12

!standard D.13.1(01)
!standard D.15(01)
!class amendment 02-06-01
!status Amendment 200Y 04-06-24
!status WG9 approved 04-11-18
!status ARG Approved 8-0-0 04-06-14
!status work item 03-12-12
!status ARG Approved 10-0-3 03-06-23
!status work item 02-06-01
!status received 02-06-01
!priority High
!difficulty Medium
!subject Timing events

!summary

A mechanism is proposed to allow user-defined procedures to be executed at a specified time without the need to use a task or a delay statement.

!problem

An exploration of various flexible scheduling schemes, for example, imprecise computation, has illustrated the need a) to asynchronously change the priority of a task at a particular future time, and b) to allow tasks to come off the delay queue at a different priority than that in effect when the task was delayed. This functionality can only be achieved by the use of a 'minder' high priority task that makes the necessary priority changes to its client. This is an inefficient and inelegant solution.

More generally, Ada provides only one mechanism for associating the execution of code with points in time. A lower level primitive would increase the applicability of the Ada language. A protected procedure can currently be associated with interrupt events; this proposal similarly allows a protected procedure to be associated with timing events.

!proposal

!wording

Add new section D.15

D.15 Timing Events

This clause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the need for a task or a delay statement.

Static Semantics

The following language-defined package exists:

```
package Ada.Real_Time.Timing_Events is

  type Timing_Event is 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 : in Time;
                         Handler : in Timing_Event_Handler);

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

  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;
```

The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event needs finalization (see 7.6).

An object of type Timing_Event is said to be *set* if it is associated with a non-null value of type Timing_Event_Handler and *cleared* otherwise.

All Timing_Event objects are initially cleared.

The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a *handler*.

Dynamic Semantics

The procedures Set_Handler associate the handler Handler with the event Event; if Handler is null, the event is cleared, otherwise it is set. The first procedure Set_Handler sets the execution time for the event to be At_Time.
The second procedure Set_Handler sets the execution time for the event to be Real_Time.Clock + In_Time.

A call of a procedure Set_Handler for an event that is already set replaces the handler and the time of execution; if Handler is not null, the event remains set.

As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event.

AARM note
The second sentence of this paragraph is because of a potential race condition. The time might expire and yet before the handler is executed, some task could call Cancel_Handler (or equivalently call Set_Handler with a null parameter) and thus clear the handler.
end AARM note

If the Ceiling_Locking policy (see D.3) is in effect when a procedure Set_Handler is called, a check is made that the ceiling priority of Handler.all is Interrupt_Priority'Last. If the check fails, Program_Error is raised.

If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past then the handler is executed immediately by the task executing the call of Set_Handler. The timing event Event is cleared.

The function Current_Handler returns the handler associated with the event Event if that event is set; otherwise it returns null.

The procedure Cancel_Handler clears the event if it is set. Cancelled is assigned True if the event was set prior to it being cleared; otherwise it is assigned False.

The function Time_Of_Event returns the time of the event if the event is set; otherwise it returns Real_Time.Time_First.

As the final step of the finalization of an object of type Timing_Event, the Timing_Event is cleared.

If several timing events are set for the same time, they are executed in FIFO order of being set.

An exception propagated from a handler invoked by a timing event has no effect.

Implementation Requirements

For a given Timing_Event object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timing_Event object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

AARM note
This prevents various race conditions. In particular it ensures that if an event occurs when Set_Handler is changing the handler then either the new or old handler is executed in response to the appropriate event. It is never possible for a new handler to be executed in response to an old event.
end AARM note

Metrics

The Implementation shall document the following metric:
An upper bound on the lateness of the execution of a handler.
That is, the maximum time between when a handler is actually executed and the time specified when the event was set.

Implementation Advice

The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.

Notes
Since a call of Set_Handler is not a blocking operation, it can be called from within a handler.

A Timing_Event_Handler can be associated with several Timing_Event objects.

Add to D.7 a new restriction identifier

No_Local_Timing_Events
Timing_Events shall be declared only at library level.

Add this restriction to the Ravenscar definition (D.13.1).

!discussion

The proposal provides an effective solution for some specific scheduling algorithms. Moreover it provides an additional paradigm for programming real-time systems. The use of timing events may reduce the number of tasks in a program and hence reduce the overheads of context switching. All tasks will suffer interference from timing events; hence they give rise to priority inversion. They are thus not an alternative to the use of time-triggered tasks, but provide an efficient means of programming short time-triggered procedures. Note, with all current implementations a low priority task coming off a delay queue causes priority inversion.

The type, Timing_Event, provides an easy means for cancelling an event. It also allows an implementation to allocate an object that will be linked into the delay queue. An implementation that links task control blocks (TCB) within its delay queue will, in effect, define a pseudo TCB for each declared timing event.

The No_Local_Timing_Events restriction is added to Ravenscar as the Ravenscar profile does not allow any local entities (those with limited lifetimes) that use the task runtime (that is, no local protected objects, no local tasks, etc.). A Timing_Event should be treated similarly.

!example

The attached appendix has an extended example but uses an earlier definition of the feature. Two further illustrations are given here. First, a watchdog timer. Here a condition is tested every 50 milliseconds. If the condition has not been called during this time an alarm handling task is released.

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;
with System; use System;
...

protected Watchdog is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Alarm_Control;    -- called by alarm handling task
  procedure Timer(Event : in out Timing_Event);    -- timer event code
  procedure Call_in;      -- called by application code every 50ms if alive
private
  Alarm : Boolean := False;
end Watchdog;

Fifty_Mil_Event : Timing_Event;
TS : Time_Span := Milliseconds(50);

protected body Watchdog is
  entry Alarm_Control when Alarm is
  begin
    Alarm := False;
  end Alarm_Control;

  procedure Timer(Event : in out Timing_Event) is
```

```
  begin
    Alarm := True;
  end Timer;

  procedure Call_in is
  begin
    Set_Handler(Fifty_Mil_Event, TS, Watchdog.Timer'Access);
    -- note, this call to Set_Handler cancels the previous call
  end Call_in;
end Watchdog;
...
```

In situations where it is necessary to undertake a small computation periodically (and with minimum jitter) the repetitive use of timing events is an effective solution. In the following example a periodic pulse is turned on and off under control of the application:

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;
with System; use System;
...

protected Pulser is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Start;
  procedure Stop;
  procedure Timer(Event : in out Timing_Event);
private
  Next_Time : Time;
end Pulser;

Pulse : Timing_Event;
Pulse_Interval : Time_Span := Milliseconds(10);


protected body Pulser is
  procedure Start is
  begin
    Pulse_Hardware;
    Next_Time := Clock + Pulse_Interval;
    Set_Handler(Pulse, Next_Time, Pulser.Timer'Access);
  end Start;

  procedure Stop is
  begin
    Cancel_Handler(Pulse);
  end Stop;

  procedure Timer(Event : in out Timing_Event) is
  begin
    Pulse_Hardware;
    Next_Time := Next_Time + Pulse_Interval;
    Set_Handler(Event, Next_Time, Pulser.Timer'Access);
  end Timer;
end Pulser;
```

!corrigendum D.7(10)

@dinsa
@xhang<@xterm<No_Asynchronous_Control>
There are no semantic dependences on the package Asynchronous_Task_Control.>
@dinst
@xhang<@xterm<No_Local_Timing_Events>
Timing_Events shall be declared only at library level.>

!corrigendum D.13.1(01)
@drepl

@xcode<@b<pragma> Restrictions (
Max_Entry_Queue_Length =@> 1,
Max_Protected_Entries =@> 1,
Max_Task_Entries =@> 0,
No_Abort_Statements,
No_Asynchronous_Control,
No_Calendar,
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Task_Allocators,
No_Task_Attributes_Package,
No_Task_Hierarchy,
No_Task_Termination,

Simple_Barriers);>
@dby
@xcode<@b<pragma> Restrictions (
Max_Entry_Queue_Length =@> 1,
Max_Protected_Entries =@> 1,
Max_Task_Entries =@> 0,
No_Abort_Statements,
No_Asynchronous_Control,
No_Calendar,
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Local_Timing_Events,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Task_Allocators,
No_Task_Attributes_Package,
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers);>

!corrigendum D.15(01)

@dinsc

This clause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the need for a task or a delay statement.

@i<@s8<Static Semantics>>

The following language-defined package exists:

```
@xcode<@b<package> Ada.Real_Time.Timing_Events @b<is>
  @b<type> Timing_Event @b<is limited private>;
  @b<type> Timing_Event_Handler
      @b<is access protected procedure> (Event : @b<in out> Timing_Event);
  @b<procedure> Set_Handler (Event   : @b<in out> Timing_Event;
                     At_Time : @b<in> Time;
                     Handler : @b<in> Timing_Event_Handler);
  @b<procedure> Set_Handler (Event   : @b<in out> Timing_Event;
                     In_Time : @b<in> Time_Span;
                     Handler : @b<in> Timing_Event_Handler);
  @b<function> Current_Handler (Event : Timing_Event)
      @b<return> Timing_Event_Handler;
  @b<procedure> Cancel_Handler (Event     : @b<in out> Timing_Event;
                       Cancelled : @b<out> Boolean);
  @b<function> Time_Of_Event (Event : Timing_Event) @b<return> Time;
@b<private>
  ... -- @ft<@i<not specified by the language>>
@b<end> Ada.Real_Time.Timing_Events;>
```

The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event needs finalization (see 7.6).

An object of type Timing_Event is said to be @i<set> if it is associated with a non-null value of type Timing_Event_Handler and @i<cleared> otherwise.
All Timing_Event objects are initially cleared.

The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a @i<handler>.

@i<@s8<Dynamic Semantics>>

The procedures Set_Handler associate the handler Handler with the event Event; if Handler is @b<null>, the event is cleared, otherwise it is set. The first procedure Set_Handler sets the execution time for the event to be At_Time.
The second procedure Set_Handler sets the execution time for the event to be Real_Time.Clock + In_Time.

A call of a procedure Set_Handler for an event that is already set replaces the handler and the time of execution; if Handler is not @b<null>, the event remains set.

As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event.

If the Ceiling_Locking policy (see D.3) is in effect when a procedure Set_Handler is called, a check is made that the ceiling priority of Handler.@b<all> is Interrupt_Priority'Last. If the check fails, Program_Error is raised.

If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past then the handler is executed immediately by the task executing the call of Set_Handler. The timing event Event is cleared.

The function Current_Handler returns the handler associated with the event Event if that event is set; otherwise it returns @b<null>.

The procedure Cancel_Handler clears the event if it is set. Cancelled is assigned True if the event was set prior to it being cleared; otherwise it is assigned False.

The function Time_Of_Event returns the time of the event if the event is set; otherwise it returns Real_Time.Time_First.

As the final step of the finalization of an object of type Timing_Event, the Timing_Event is cleared.

If several timing events are set for the same time, they are executed in FIFO order of being set.

An exception propagated from a handler invoked by a timing event has no effect.

@i<@s8<Implementation Requirements>>

For a given Timing_Event object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timing_Event object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

@i<@s8<Metrics>>

The implementation shall document the following metric: @xbullet<An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set.>

@i<@s8<Implementation Advice>>

The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.

@xindent<@s9<NOTES@hr
Since a call of Set_Handler is not a blocking operation, it can be called from within a handler.>>

@xindent<@s9<A Timing_Event_Handler can be associated with several Timing_Event objects.>>

! ACATS Test

ACATS test(s) need to be created.

!appendix

!standard D.8
!class amendment
!status

!priority -- we consider this to be high
!difficulty
!subject Timing Events
!from A. Burns and A.J. Wellings on behalf of IRTAW11

! summary

A mechanism is proposed to allow user-defined procedures to be executed at a specified time without the need to use a task or a delay statement.

! problem

An exploration of various flexible scheduling schemes, for example imprecise computation, has illustrated the need a) to asynchronously change the priority of a task at a particular future time, and b) to allow tasks to come off the delay queue at a different priority to that in effect when the task was delayed. This functionality can only be achieved by the use of a 'minder' high priority task that makes the necessary priority changes to its client. This is an inefficient and inelegant solution.

More generally, Ada provides only one mechanism for associating the execution of code with points in time. A lower level primitive would increase the applicability of the Ada language. A protected procedure can currently be associated with interrupt events; the proposal allows similar functionality for timing events.

! proposal

A child package of Ada.Real_Time is proposed.

```
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Real_Time.Timing_Events is

  type Timing_Event is limited private;

  type Parameterless_Handler is access protected procedure;

  procedure Set_Handler(TE : in out Timing_Event; At_Time : Time;
            Handler: Parameterless_Handler);

  procedure Set_Handler(TE : in out Timing_Event; In_Time: Time_Span;
            Handler: Parameterless_Handler);

  function Is_Handler_Set(TE : Timing_Event) return boolean;

  procedure Cancel_Handler(TE : in out Timing_Event);

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

A call to a Set_Handler procedure returns when the Handler is registered.

At a time no earlier than that implied by the time parameter, the handler procedure is executed. The detailed rules governing timing accuracy are the same as D.9. The assumption is that the procedure may be executed by the real-time clock interrupt mechanism. Hence the ceiling priority of the protected procedure (Parameterless_Handler) must be interrupt_priority'last. It is a bounded error to use a protected procedure with a priority less than interrupt_priority'last.

If a Set_Handler procedure is called with zero or negative In_Time or with At_time indicating a time in the past then the Handler is executed immediately by the task executing the Set_Handler call. The rationale for this follows from a comparison with delay and delay until that act as null operations in these circumstances. An exception raised during the execution of a Handler is ignored (c.f. interrupts - C.3(7)).

A call of either Set_Handler procedure is not a potentially suspending operation and hence can be called from within a Handler (e.g. a Handler can set up a future timing event).

A call to a Set_Handler procedure for a timing event that is already set will override the first set operation (i.e. equivalent to making a call to Cancel_Handle first).

A call to Is_Handler_Set following a call to Cancel_Handler for the same timing_event will return false. A call to Cancel_Handler will have no effect if the timing event is not set.

A number of timing events registered for the same time will execute in FIFO order. Note, all will be executed before any other application code.

! discussion

The proposal provides an effective solution for some specific scheduling algorithms. Moreover it provides an additional paradigm for programming real-time systems. The use of timing events may reduce the number of tasks in a program and hence reduce the overheads with context switching. All tasks will suffer interference from timing events; hence they give rise to priority inversion. They are thus not an alternative to the use of time-triggered tasks, but provide an efficient means of programming short time-triggered procedures. Note, with all current implementation a low priority task coming off a delay queue will cause priority inversion.

The type, timing_event, provides an easy means for cancelling an event. It also allows an implementation to allocate an object that will be linked into the delay queue. An implementation that links task control blocks (TCB) within its delay queue will, in effect, define a pseudo TCB for each declared timing event.

! examples

The attached appendix has an extended example. Two further illustrations are given here. First a watchdog timer. Here a condition is tested every 50 milliseconds. If the condition has not been called during this time an alarm handling task is released.

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;
with System; use System;
...

protected Watchdog is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Alarm_Control;    -- called by alarm handling task
```

```
  procedure Timer;        -- timer event code
  procedure Call_in;      -- called by application code every 50ms if alive
private
  Alarm : Boolean := False;
end Watchdog;

Fifty_Mil_Event : Timing_Event;
TS : Time_Span := Milliseconds(50);

protected body Watchdog is
  entry Alarm_Control when Alarm is
  begin
    Alarm := False;
  end Alarm_Control;

  procedure Timer is
  begin
    Alarm := True;
  end Timer;

  procedure Call_in is
  begin
    Set_Handler(Fifty_Mil_Event, TS, Watchdog.Timer'access);
    -- note, this call to Set_Handler cancels the previous call
  end Call_in;
end Watchdog;
...
```

In situations where it is necessary to undertake a small computation periodically (and with minimum jitter) the repetitive use of timing events is an effective solution. In the following example a periodic pulse is turned on and off under control of the application:

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;
with System; use System;
...

protected Pulser is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Start;
  procedure Stop;
  procedure Timer;
private
  Next_Time : Time;
end Pulser;

Pulse : Timing_Event;
Pulse_Interval : Time_Span := Milliseconds(10);

protected body Pulser is
  procedure Start is
  begin
    Pulse_Hardware;
    Next_Time := Clock + Pulse_Interval;
    Set_Handler(Pulse, Next_Time, Pulser.Timer'access);
```

```
  end Start;

  procedure Stop is
  begin
    Cancel_Handler(Pulse);
  end Stop;

  procedure Timer is
  begin
    Pulse_Hardware;
    Next_Time := Next_Time + Pulse_Interval;
    Set_Handler(Pulse, Next_Time, Pulser.Timer'access);
  end Timer;
end Pulser;
```

! appendix.

The following is taken from a paper by Burns and Wellings at IRTAW11.

Accessing Delay Queues

A. Burns and A.J. Wellings
Real-Time Systems Research Group Department of
Computer Science, University of York, UK

Abstract
A number of flexible scheduling schemes can be programmed in Ada 95 by combining the more advanced features the language provides. For example, imprecise computations would appear to be accommodated by the use of ATCs (with timing triggers) and dynamic priorities. Unfortunately with this type of scheme, it is difficult to ensure that the priority of the task, following the triggering event, is at the appropriate level. This problem is investigated and a potential solution is described. It involves opening up the implementation of the delay queue so that application code can be executed directly when the time is right.

1 Introduction

The dynamic priority facility of the Real-Time Systems Annex provides a flexible means of deriving alternative scheduling algorithms (from the predefined fixed priority preemptive approach). For example, it is relatively easy to code an Earliest Deadline scheme with this provision [3]. In a recent paper [1] we showed how a powerful scheduling framework can be implemented in Ada via the use of a combination of advanced tasking features, including dynamic priorities. Unfortunately in many of these flexible scheduling schemes it is necessary to asynchronous change the priority of a task. This is either to come off a delay queue at a different priority to that which was in operation when the delay request was made or to move from a low priority to a higher one at a specified point in time. In both of these circumstances it is impossible for the task itself to change its own priority; it is either not runnable (i.e. delayed) or is potentially not running (as its low priority is not high enough - this is the reason that a priority switch is being undertaken). Hence the flexible algorithms that have been published require a supportive (minder) task that:

o runs at the right time,
o runs at the high priority, and

o dynamically raises the priority of its client task

Although this is adequate, in the sense of delivering the correct behaviour, it is inefficient and inelegant. In the worst case, if all tasks wish to exhibit flexibility, the number of tasks in the program will be doubled. (It is possible to program a single minder task to deal with all priority changes for all tasks, but this minder task must contain the equivalent of its own delay queue which may be inefficient.)

The reason for Ada's difficulty with these algorithms is that only one kind of software entity can wait on a timing event. Only the task can execute a delay. In this paper we consider a means by which the implementation of the delay queue can be opened up so that a protected procedure can be executed directly when a specified time is reached. The proposal is given in section 3, following an illustrative example in the next section. Section 4 then considers the implications for the delay and delay until statements themselves. Ravenscar issues are discussed briefly in section 5 and conclusions are given in section 6.

2 Imprecise Computation - An Example of the use of Dynamic Priorities

In our 1997 Workshop paper [2] we discussed the following example. One means of increasing the utilisation and effectiveness of real-time applications is to use the techniques that are known, collectively, as imprecise computations [5, 4]. Tasks are structures into two phases, a mandatory part that, as its name suggests, must be executed; and an optional part.

Various scheduling approaches are used to try and increase the likelihood that optional parts are completed, but they do not have to be guaranteed. With fixed priority scheduling, the mandatory phases are assigned priorities using the deadline monotonic or rate monotonic algorithms. The optional parts are assigned lower priorities (than any mandatory value). It is not the concern of this paper to discuss how these priorities are obtained (or if they are static or themselves dynamic).

The following code gives a typical periodic task with mandatory and optional phases. It has a period of 50ms and a deadline of 40ms. In the mandatory phase, an adequate output value (Result) is computed and stored (externally) in a simple protected object (Store).
During the optional part, more precise values of Result are computed and stored. The optional part is abandoned (using an ATC) when the task's deadline arrives.

```
with Ada.Real_Time;
with Ada.Task_Identification;
with Ada.Dynamic_Priorities;
with System;
...

Mandatory_Pri : constant System.Priority := ...;
Optional_Pri :  constant  System.Priority := ...;
              -- less than mandatory

protected Store is
 procedure Put(X : Some_Data_Type);
 procedure Get(X : out Some_Data_Type);
private
...
end Store;

protected body Store is ...

task Example is
```

```
  pragma Priority(Mandatory_Pri);
end Example;

task body Example is
  Start_Time : Ada.Real_Time.Time := Ada.Real_Time.Clock;
  Period : Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(50);
  Deadline : Ada.Real_Time.Time_Span:= Ada.Real_Time.Milliseconds(40);
  Result : Some_Data_Type;
begin
  loop
    -- code of the mandatory part, including
    Result := ...
    Store.Put(Result);
    select
      delay until Start_Time + Deadline;
      Ada.Dynamic_Priorities.Set_Priority(Mandatory_Pri);
    then abort
      Ada.Dynamic_Priorities.Set_Priority(Optional_Pri);
      loop
        -- code of the optional part, including
            Result := ... Store.Put(Result);
      end loop;
    end select;
    Start_Time := Start_Time + Period;
    delay until Start_Time;
  end loop;
end Example;
```

Unfortunately this code is not correct. The task starts the ATC delay with a high priority. In the abortable region the priority is lowered. When the timeout occurs the task may not be executing and hence it may never get to raise its priority back to the right level for its next invocation. Even if we use finalisation in the abortable part there is no guarantee that the finalisation code will be executed as this will also occur at the lower priority.

2.1 Possible Work-Arounds

This approach uses a shadow task that always runs at the mandatory priority. Its sole job is to raise the priority of the `real' task when its deadline is due. To make sure the tasks execute in a coordinated way, the start time of the `client' task is passed to the minder task using a rendezvous. For one task to change the priority of another requires the task ID to be known. We capture this

during the simple rendezvous.

```
task Minder is
  entry Register(Start : Ada.Real_Time.Time);
  pragma Priority(Mandatory_Pri);
end Minder;

task Example is
  pragma Priority(Mandatory_Pri);
end Example;

task body Example is
  Start_Time : Ada.Real_Time.Time := Ada.Real_Time.Clock;
  Period : Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(50);
```

```
  Deadline : Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(40);
  Result : Some_Data_Type;
begin
  Minder.Register(Start_Time);
  ... As Before But No Call To Increase Priority
end Example;

task body Minder is
  Start_Time : Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span:= Ada.Real_Time..Milliseconds(50);
  Offset : Ada.Real_Time.Time_Span:= Ada.Real_Time.Milliseconds(40);
  Id : Ada.Task_Identification.Task_Id;
begin
  accept Register(Start : Ada.Real_Time.Time) do
    Id := Register'Caller;
    Start_Time := Start;
  end Register;
  Start_Time := Start_Time + Offset;
  loop
    delay until Start_Time;
    Ada.Dynamic_Priorities.Set_Priority(Mandatory_Pri,Id);
    Start_Time := Start_Time + Period;
  end loop;
end Minder;
```

3 Proposed Alternative Scheme for Executing Timely Code

To remove the need for the extra minder task, we draw an analogy with Ada's interrupt handling provisions. For non-timing events (interrupts) it is possible to attach code (a parameter less protected procedure) to the event in such a way that the code is executed when the event (interrupt) occurs. The system clock reaching a specified time can also be seen as an event. Indeed delays queues are events `waiting to happen' that are implemented by managing interrupts from the system clock.

In this paper we shall consider `time' as defined in Annex D and hence the most natural place to add the provision of a time-based signaling facility is a child package of Ada. Real Time:

```
package Ada.Real_Time.Timing_Events is
  type Parameterless_Handler is access protected procedure;

  procedure Signal(At_Time : Time; Handler : Parameterless_Handler);
  procedure Signal(In_Time : Time_Span; Handler : Parameterless_Handler);

end Ada.Real_Time.Timing_Events;
```

Two routines are required due to the usual need to express absolute and relative times. A call of Signal will return once the handler request is registered. When the time indicated is reached (or the interval of time expired) then the protected procedure associated with the handler will be executed. As with the definition of the delay statements, clock granularity issues may mean that the handler will be executed after the indicated time – it will never be execute before.

The definition allows more than one call to Signal from the same task. Hence more than one handler may need to be executed at the same time instance. The order in which an implementation will execute these handlers is not defined. Indeed the same handler may be registered on a number of occasions for different (or indeed the same) time instance. We have decided not to include a means of canceling a future timing event. The overhead is allowing

cancellation is likely to be high. In terms of the API, the above definitions would probably have to be modified so that calls to Signal would return some form of ID that could then be used to cancel the event.

With any interrupt handler it is necessary to assign the correct ceiling priority to the protected object that contains the handler. This is also the case with these timing events. As the clock interrupt is often the highest in the system we assume Interrupt Priority' Last will be required as the ceiling. Note also that the protected object must be defined at the library level to ensure it is visible.

3.1 Imprecise Computation Example Revisited

With this new facility the example given earlier is easily accommodated. Only a single task is required (together with an appropriate protected object).

```
protected type Minder is
  pragma Priority Interrupt_Priority'Last;
  procedure Change; procedure Register;
private
  Id : Ada.Task_Identification.Task_Id;
end Minder;

protected body Minder is
  procedure Register is
  begin
    Id := Ada.Task_Identification.Current_Task;
  end Register;

  procedure Change is
  begin
    Ada.Dynamic_Priorities.Set_Priority(Mandatory_Pri,Id);
  end Change;
end Minder;

task body Example is
  Start_Time : Ada.Real_Time.Time:= Ada.Real_Time.Clock;
  Period : Ada.Real_Time.Time_Span:= Ada.Real_Time.Milliseconds(50);
  Deadline : Ada.Real_Time.Time_Span:= Ada.Real_Time.Milliseconds(40);
  Result : Some_Data_Type;
  Pri_Control : Minder;
begin
  Pri_Control.Register;
  loop
    -- code of the mandatory part, including
    Result := ...
    Store.Put(Result);
    Ada.Real_Time.Timing_Events.Signal(Start_Time + Deadline,
                                       Pri_Control.Change);
    select
      delay until Start_Time + Deadline;
    then abort
      Ada.Dynamic_Priorities.Set_Priority(Optional_Pri);
      loop
        -- code of the optional part, including
           Result := ...
           Store.Put(Result);
```

```
            -- note, no exit statement
      end loop;
    end select;
    Start_Time := Start_Time + Period;
    delay until Start_Time;
  end loop;
end Example;
```

3.2 Scheduling Implications

From a scheduling standpoint the use of timing events has two implications:

o The number of task is reduced and the cost of context switching to minder tasks is eliminated.

o All tasks suffer interference from all timing events.

This clearly means that an application must consider the trade-off that the facility provides. If the timing event handlers are long then it would be better to encapsulate them in a task that can execute at the right priority. But if they are short then then may as well be executed as part of the clock interrupt handler that will happen anyway. Remember there is no easy means of stopping all tasks suffering interference from all delay expirations.

4 Delay Statements Revisited

By introducing timing events we are in effect opening up the implementation of the delay mechanisms. A process known as reflection. Indeed once we have timing events then a programmer can construct delay statements:

```
package Delay_Routines is
  -- routines for just one task
  procedure Del(Int : Time_Span);
    -- identical to delay
  procedure Del_Til(T : Time);
    -- identical to delay until
end Delay_Routines;

package body Delay_Routines is
  protected Delayer is
    pragma Priority Interrupt_Priority'Last;
    entry Del;
    procedure Now;
  private
    Continue : Boolean := False;
  end Delayer;

  protected body Delayer is
    entry Del when Continue is
    begin
      Continue := False;
    end Del;
    procedure Now is begin
      Continue := True;
    end Now;
  end Delayer;

  procedure Del(Int : Time_Span) is
```

```
  begin
    Ada.Real_Time.Timing_Events.Signal(Int,Delayer.Now);
    Delayer.Del;
  end Del;

  procedure Del_Til(T : Time) is
  begin
    Ada.Real_Time.Timing_Events.Signal(T,Delayer.Now);
    Delayer.Del;
  end Del_Til;

end Delay_Routines;
```

In the example given earlier in section 3.1, the return from the delay statement and the signal were both programmed to occur at the same time. A more efficient implementation would combine them:

```
protected type Minder is
  pragma Priority Interrupt_Priority'Last;
  procedure Change;
  procedure Register;
  entry Wake_Up;
private
  Id : Ada.Task_Identification.Task_Id;
  Optional_Over : Boolean := False;
end Minder;

protected body Minder is
  procedure Register is
  begin
    Id := Ada.Task_Identification.Current_Task;
  end Register;

  entry Wake_Up when Optional_Over is
  begin
    Optional_Over := False;
  end Wake_Up;

  procedure Change is begin
    Ada.Dynamic_Priorities.Set_Priority(Mandatory_Pri,Id);
    Optional_Over := True;
  end Change;
end Minder;

task body Example is
  Start_Time : Ada.Real_Time.Time:= Ada.Real_Time.Clock;
  Period : Ada.Real_Time.Time_Span:= Ada.Real_Time.Milliseconds(50);
  Deadline : Ada.Real_Time.Time_Span:= Ada.Real_Time.Milliseconds(40);
  Result : Some_Data_Type;
  Pri_Control : Minder;
begin
  Pri_Control.Register;
  loop
    -- code of the mandatory part, including
    Result := ...
    Store.Put(Result);
```

```
   Ada.Real_Time.Timing_Events.Signal(Start_Time + Deadline,
                                      Pri_Control.Change);
   select
     Pri_Control.Wake_Up;
   then abort
     Ada.Dynamic_Priorities.Set_Priority(Optional_Pri);
     loop
       -- code of the optional part, including
           Result := ...
           Store.Put(Result);
           -- note, no exit statement
     end loop;
   end select;
   Start_Time := Start_Time + Period;
   delay until Start_Time;
  end loop;
end Example;
```

5 Ravenscar Issues

At the last Workshop extensions to Ravenscar were discussed [6]. One observation coming from some of the position papers was that Ravenscar plus Dynamic Priorities represents a powerful set of facilities. In this paper we have motivated the need for timing events by using the full set of tasking features. However, the combination of Ravenscar plus Dynamic Priorities plus timing events is a profile worthy of further evaluation.

6 Conclusions

Ada 95 contains a very flexible concurrency model and many real-time features. This makes it the most general purpose real-time programming language in common usage. Indeed the combination of ATCs and the use of dynamic priorities allow a wide range of scheduling schemes to be supported. Unfortunately in many of these schemes extra minder tasks are needed to manipulate priorities at designated times.

Arguable a real-time programming language should provide more than one means of bringing time and code together. In this paper we have investigated the use of timing events that are execute directly by the run-time system. This appears to be a powerful general purpose language primitive which Ada implementations are able to provide with little effort.

References

[1] A. Burns and G. Bernat. Implementing a flexible scheduler in Ada. In Reliable Software Technologies, Proceedings of the Ada Europe Conference, Leuven, pages 179 - 190. Springer Verlag, LNCS 2043, 2001.

[2] A. Burns and A.J. Welling. Feature interaction with dynamic priorities. In A.J. Wellings, editor, Proceedings of the 8th International Real-Time Ada Workshop, pages 27-32. ACM Ada Letters, 1997.

[3] A. Burns and A. J. Wellings. Concurrency in Ada. Cambridge University Press, 1995.

[4] J.W.S. Liu, K.J. Lin, W.K. Shih, AC. S. Yu, J.Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. EEE Computer, pages 58-68, 1991.

[5] W. K. Shih, J. W. S. Liu, and J. Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. In Proc. IEEE Real-Time Systems Symposium, 1989.

[6] A.J. Wellings. Status and future of the Ravenscar profile: Session summary. In M.G. Harbour, editor, Proceedings of the 10th International Real-Time Ada Workshop, pages 5-8. ACM Ada Letters, 2001.

****

From: Tucker Taft
Sent: Thursday, September 5, 2002 9:56 AM

Alan Burns wrote:

```
> ...
> !proposal
>
> A child package of Ada.Real_Time is proposed.
>
> package Ada.Real_Time.Timing_Events is
>
>   type Timing_Event is limited private;
>
>   type Timing_Event_Handler(TE : in out Timing_Event)
>     is access protected procedure;
```

This should be:

```
   type Timing_Event_Handler is
     access protected procedure(TE: in out Timing_Event);
```

```
>
>   procedure Set_Handler(TE : in out Timing_Event; At_Time : Time;
>             Handler: Timing_Event_Handler);
>
>   procedure Set_Handler(TE : in out Timing_Event; In_Time: Time_Span;
>             Handler: Timing_Event_Handler);
```

Once you give a mode for one parameter, it seems to be "good form" to use explicit rather than implicit "in" thereafter. Hence, I would recommend adding "in" after "*_Time:" and "Handler:".

```
>
>   function Is_Handler_Set(TE : Timing_Event) return boolean;
>
>   function Current_Handler(TE : Timing_Event) return Timing_Event_Handler;
>
>   procedure Cancel_Handler(TE : in out Timing_Event);
>
>   function Time_Of_Event(TE : Timing_Event) return Time;
>
> private
>   ... -- not specified by the language
> end Ada.Real_Time.Timing_Events;
> ...
```

****

From: Alan Burns
Sent: Thursday, September 5, 2002 10:09 AM

Thanks for stopping silly mistake with the parameter.

I'll note you advice about in parameters

****

From: Robert A. Duff
Sent: Thursday, September 5, 2002 5:38 PM

> Once you give a mode for one parameter, it seems to be "good form" to use explicit rather than
> implicit "in" thereafter.
> Hence, I would recommend adding "in" after "*_Time:" and "Handler:".

For what it's worth, I don't agree with that advice. My advice is:
Never use 'in'; always make it implicit.

I don't see the point in Tucker's advice on this point.

****

From: Robert Dewar
Sent: Thursday, September 5, 2002 7:10 PM

I strongly agree with Bob here, I find gratuitous IN mode indications (such as those that infest the RM) annoying :-)

****

From: Tucker Taft
Sent: Thursday, September 5, 2002 9:07 PM

I think consistency with the rest of the RM is appropriate. The general rule in the RM seems to be *always* use explicit parameter modes on procedures. For functions, either always or never in a given package.

I don't believe we should be debating our own personal styles here, but rather the de-facto "RM style." That's what I meant by "good form."

****

From: Robert A. Duff
Sent: Friday, September 6, 2002 7:39 PM

OK, fine. You confused me by saying it has something to do with the
previous parameter, which is a rule I had never heard of.

****

From: Robert Dewar
Sent: Friday, September 6, 2002 4:28 PM

<<I don't believe we should be debating our own personal styles here, but rather the de-facto "RM style." That's what I meant by "good form.">>

Yes, that makes sense, but it means that pushing a personal style of giving modes once one mode has been given is also inappropriate :-)

****

From: Tucker Taft
Sent: Friday, September 6, 2002 8:59 PM

Yes, I agree. I was really making an imperfect guess what was the RM style. I took a closer look afterward and discovered that we always used modes for procedures, and always or never used them for functions in a given package.

****

From: Robert Dewar
Sent: Friday, September 6, 2002 9:25 PM

For the record, the reason I prefer to leave out IN (and object to the implied recommendation in the RM) is that IN parameters are indeed the normal cases. OUT and IN OUT parameters are the exceptional cases, and I prefer to use the keywords to point out the exceptional cases.

****

From: Randy Brukardt
Sent: Thursday, January 2, 2003 4:22 PM

In finishing up the minutes, I came across the discussion about ways to force these to be declared at the library level. We dreamt up a complex and dangerous mechanism to handle that in AI-303. In thinking about that a bit, I wonder if we aren't going about this entirely the wrong way.

Finalization is a core capability in Ada 95. It is used to perform all sorts of tasks when objects go out of scope. For example, the canonical semantics for protected objects uses finalization when they are destroyed.

Some real-time people claim that finalization is too expensive for their applications. This is a rare special case (finalization takes very little overhead, far less than a rendezvous, so this applies only to the most critical applications). Thus, Ada provides ways to eliminate the overhead. For example, the Ravenscar profile adds a Restriction "No_Local_Protected_Objects" to eliminate this overhead. Why can't we use a similar solution for timing objects?

That is, define the finalization of timing objects to unhook them properly.
Then, add a restriction "No_Local_Timing_Objects" to eliminate the (small) overhead of finalization for critical real-time applications. (Presumably, this restriction would be added to the Ravenscar profile).

Since we've decided to use a package-based approach for timing object, this isn't even going to be very expensive to implement: a canonical timing object can be derived from Limited_Controlled in the private part with a private overriding Finalize. Then the existing language will do all of the hard work. Ravenscar runtimes presumably would do something different (depending on the restriction), but they typically are completely different than a full runtime anyway.

****

From: Alan Burns
Sent: Saturday, November 8, 2003 3:13 AM

A minor issue was raised by someone at the IRTAW reading the AI on timing events. I would hope we could deal with this during editorial review.

Under the dynamic semantics we have:

>>
Following a call of a Set_Handler procedure, at a time no earlier than that specified by the time parameter, the Timing_Event_Handler procedure is executed. The rules governing timing accuracy are the same as in D.9.

The Timing_Event_Handler is only executed if the timing event is set at the time of execution. Following the execution of the Timing_Event_Handler the Timing_Event denoted by Event is cleared.
<<

The last sentence is the issue.

A common use of timing events will be to reset the event for some time in the future while handling the current event. The semantics are therefore more straightforward if the Event is cleared as the initial action of executing the handler (allowing it to be set again during the execution of the handler). The current last sentence implies clearing could happen as the final act which is not what was intended.

I would propose changing the last sentence to:

>>
As the initial action of the execution of the
Timing_Event_Handler the Timing_Event denoted by Event is cleared.
<<

This makes is it clear.

Acceptable?

****

From: Erhard Ploedereder
Sent: Sunday, November 23, 2003 4:46 PM

I have some major editorial problems with the! wording. For example:

The last paragraph of the static semantics talks about something that is clearly dynamic semantics.

Nowhere do I find words that really explain the semantics of Set_Handler (e.g., that it registers the handler). There is talk of a time parameter, but there are two versions, one using type Time, the other Time_Span; the difference is never explained in words.

The term "can be called from within an object of type X" doesn't mean anything in LRM-ese, especially when X is an access type.

There is a bit of a contradiction on the negative time rule, which specifies that the handler gets executed by the calling task, while elsewhere (far away), the semantics say that the call returns after the Timing Event is set. I hate to have to gather the semantics of an interface piecemeal from 3 different places in the section.

In short, while the intent of the AI looks o.k., the wording needs work to match LM style of presenting packages.

****

From: Alan Burns
Sent: Monday, November 24, 2003 5:56 PM

As this is 'wording' I'll leave the detailed changes up to Randy - but here are some observations

>
> The last paragraph of the static semantics talks about something that is clearly dynamic semantics.

agreed

>
> Nowhere do I find words that really explain the semantics of Set_Handler (e.g., that it registers the handler). There is talk of a time parameter, but there are two versions, one using type Time, the other Time_Span; the difference is never explained in words.

The first words after the package do clearly imply an object of type Timing_Event is set if it has a registered Timing_Event_Handler. Is it not clear that Time is absolute and Time_Span relative? and both define the time for the event to fire

>
> The term "can be called from within an object of type X" doesn't mean anything in LRM-ese, especially when X is an access type.

yes, I'll leave that to expert LRM-ese editor

>
> There is a bit of a contradiction on the negative time rule, which specifies that the handler gets executed by the calling task, while elsewhere (far away), the semantics say that the call returns after the Timing Event is set. I hate to have to gather the semantics of an interface piecemeal from 3 different places in the section.

There are a number of points to make about the semantics; they cannot all be in the same paragraph. As it is the normal behaviours are closer together. But obviously others (editors for example) may feel a different order would be preferred.

****

From: Randy Brukardt
Sent: Monday, November 24, 2003 8:34 PM

Alan said, responding to Erhard:

> > The term "can be called from within an object of type X" doesn't
> > mean anything in LRM-ese, especially when X is an access type.
>

> yes, I'll leave that to expert LRM-ese editor

Gee, thanks. :-)

The entire "can be called part" seems to be a note; it isn't normative in any way. I'd either put it in the AARM or as a RM note; either way, I don't have to rewrite it (other than to add "designated"). Any preference on which?

****

From: Alan Burns
Sent: Tuesday, November 25, 2003 2:36 AM

I agree this isn't normative. I would prefer to see a RM note.

****

From: Robert Dewar
Sent: Tuesday, November 25, 2003 6:01 AM

It should be an RM note. The AARM has no formal status, and should not be used for any important information.

****

From: Robert A. Duff
Sent: Sunday, November 30, 2003 5:34 PM

Neither RM notes nor AARM annotations have "normative" status under ISO rules. During the Ada 9X project, the primary criterion for choosing between the two was that RM notes should be of use to Ada programmers, whereas AARM annotations should be of use to compiler writers and language designers.

Sorry for being overly emotional, but I take a little bit of offense at the implication that AARM annotations are not "important information".

I'm a bit touchy on the subject, having expended a lot of energy trying to put information useful, and even "important", to compiler writers and language designers into the AARM! ;-)

****

From: Robert Dewar
Sent: Sunday, November 30, 2003 10:47 PM

That's a very reasonable viewpoint, but all too often information ends up in the AARM which is indeed useful to Ada programmers. And I think this particular case is an example.

****

From: Robert A. Duff
Sent: Monday, December 1, 2003 8:12 AM

Yeah, there was a lot of pressure to keep the RM as small as possible, so we probably did put some things in the AARM when an RM note would have been more appropriate.

****

From: Alan Burns
Sent: Wednesday, December 17, 2003 11:15 AM

AI 297 was returned by WG9 to the ARG.
There was a number of wording comments, Randy dealt with some of these but felt I should look at a couple. First the meaning of the parameters to Set_Handler was not explicit. I've added a couple of sentences to Randy's new first paragraph of dynamic semantics - remember the model is that the event is set when a handler is registered.

The other point concerned being explicit about the time at which the handler is executed. Randy felt reference to D.9 did not help. Also it was felt that saying handler executed after some defined time was not enough (ie no upper bound). I've looked at the words for delay statements and interestingly they do not give an upper bound either, but give a metric. So I've chosen to use the words suggested 'As soon as possible after' and added a metric, but removed reference to D.9.

I've reordered a couple of other paragraphs to follow suggestion to use order in package to order these paragraphs.

I'll give the wording section only as I hope those concerned with the wording of the AI can check that the changes are appropriate. It would be good if this can be done soon.
Wording can then be placed back in the full AI (by Randy?)

!wording

Add new section D.15

D.15 Timing Events

This clause introduces a language-defined child package of Ada.Real_Time to allow user-defined protected procedures to be executed at a specified time without the need to use a task or a delay statement.

Static Semantics

The following language-defined package exists:

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is 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 : in Time; Handler: in Timing_Event_Handler);
  procedure Set_Handler(Event : in out Timing_Event;
            In_Time: in Time_Span; Handler: in 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;
```

An object of type Timing_Event is said to be set if it has a registered Timing_Event_Handler. An object is said to be cleared if it has no registered Timing_Event_Handler. All Timing_Event objects are initially cleared.

Dynamic Semantics

A call to a Set_Handler procedure returns after the Timing_Event_Handler denoted by Handler is registered. The first Set_Handler procedure registers the Timing_Event_Handler for execution at time At_Time. The second Set_Handler procedure registers the Timing_Event_Handler for execution at time Ada.Real_Time.Clock + In_Time;

A call to a Set_Handler procedure for a Timing_Event that is already set will initially cancel the earlier registration. The Timing_Event denoted by Event remains set.

A call of either Set_Handler procedure is not a potentially blocking operation.

As soon as possible after the time registered for the event, the Timing_Event_Handler procedure is executed. The Timing_Event_Handler is only executed if the timing event is set at the time of execution. As the initial action of the execution of the Timing_Event_Handler the Timing_Event denoted by Event is cleared.

If the Ceiling_Locking policy (see D.3) is in effect when a Set_Handler procedure is called, a check is made that the ceiling priority of Timing_Event_Handler is Interrupt_Priority'last. If the check fails, Program_Error is raised.

If a Set_Handler procedure is called with zero or negative In_Time or with At_Time indicating a time in the past then Timing_Event_Handler is executed immediately by the task executing the Set_Handler call. The Timing_Event denoted by Event is cleared and the handler is not registered.

An exception propagated from a Timing_Event_Handler invoked by a timing event has no effect.

A call to Is_Handler_Set returns True if Event is set; otherwise it returns False.

A call to Current_Handler returns with the current Timing_Event_Handler. If the Timing_Event denoted by Event is not set, Current_Handler returns null.

A call to Cancel_Handler returns after the Timing_Event denoted by Event is cleared. Cancelled is assigned True if Event was set prior to it being cleared; otherwise the parameter is assigned False.

A call to Time_Of_Event returns with the time of Event.
If Event is not set, Time_Of_Event returns Ada.Real_Time.Time_First.

As the final step of finalization of an object of type Timing_Event, the Timing_Event is cleared.

If several timing events are registered for the same time, they are executed in FIFO order of registration.

Metrics
The Implementation shall document the following metric:
An upper bound on the lateness of the execution of a registered handler.

That is, the maximum time between when a handler is actually executed and the time specified in the registration of that handler.

Implementation Advice
The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.

Notes
Since a call to Set_Handling is a not a blocking operation, it can be called from within an object of type Timing_Event_Handler.

Add to D.7 a new restriction identifier

No_Local_Timing_Events
Timing_Events shall be declared only at library level.

Add this restriction to the Ravenscar definition (D.13).

****

From: Robert A. Duff
Sent: Wednesday, December 17, 2003 2:20 PM

The other point concerned being explicit about the time at which the handler is executed. Randy felt reference to D.9 did not help. Also it was felt that saying handler executed after some defined time was not enough (ie no upper bound). I've looked at the words for delay statements and interestingly they do not give an upper bound either, but give a metric. So I've chosen to use the words suggested 'As soon as possible after' and added a metric, but removed reference to D.9.

I think you had it right the first time. Those "metrics" are completely meaningless, yet they pretend to be requiring something of the implementation. Useless verbiage. The better way to express this sort of thing is with Implementation Advice (which you did).

****

From: Tucker Taft
Sent: Wednesday, December 17, 2003 5:05 PM

Those "metrics" are completely meaningless, yet they pretend to be requiring something of the implementation. Useless verbiage.

I'm not sure I agree. It seems clear that in the real-time community, you really do want the vendor to provide metrics (or at least feel a bit guilty if they don't).

The better way to express this sort of thing is with Implementation Advice (which you did).

Implementation advice is also useful, but clearly serves a somewhat different purpose. Neither has any real weight. Both can be useful in establishing some "norms."

****

From: Robert Dewar
Sent: Wednesday, December 17, 2003 11:15 AM

Tucker Taft wrote:

>>... Those "metrics" are completely
>>meaningless, yet they pretend to be requiring something of theimplementation. Useless verbiage.

>
>
> I'm not sure I agree. It seems clear that in the real-time community, you really do want the vendor to provide metrics (or at least feel a bit guilty if they don't).

As normative requirements, these are complete junk, since they do not define the form or requirements sufficiently well and use undefined terms.

>>... The better way to express this sort
>>of thing is with Implementation Advice (which you did).
>
> Implementation advice is also useful, but clearly serves a somewhat different purpose. Neither has any real weight. Both can be useful in establishing some "norms."

I actually think that implementation advice ends up being stronger here. First, we can say what we want, without worrying whether it has formal semantic meaning. Second, we expect at least documentation of what is not supported in IA (and I think it would be reasonable to require the reason for not following any IA in the documentation).

****

From: Tucker Taft
Sent: Thursday, December 18, 2003 9:18 AM

In the spirit of talking with "real users", perhaps we should ask the real-time user community rather than vendors whether these "documentation requirements" have turned out to be of any use. I personally have no idea.

The value of implementation advice seems to be in its guidance to implementers and in its documentation requirement. It would seem that metrics have exactly the same value, namely guidance in the sense of what time bounds are relevant to customers, and documentation requirements in terms of what numbers to provide (and perhaps the process of getting them is also useful activity).

The question is which documentation is actually provided by vendors, and then which documentation is actually of value to users.

My hope is that both are provided, and both are of value. But I have been known to be an optimist ;-).

****

From: Robert A. Duff
Sent: Thursday, December 18, 2003 12:04 PM

> In the spirit of talking with "real users", perhaps we should
> ask the real-time user community rather than vendors whether these
> "documentation requirements" have turned out to be of any use.
> I personally have no idea.

That's certainly a good idea, but one of the things I don't like about the Metrics is that because of their style, they might trick non-language-lawyers into thinking they actually formally *require* something. We might find programmers who say, "of course we want to know about interrupt-handling overhead", but if that information is bogus, well, ...

It seems to me that Implementation Advice along the lines of "... should take a small and bounded amount of time" would be more informative.

As *requirements*, the Metrics are silly. Real-time programmers use integer addition more often than they use delay statements, and they want bounded-time integer addition. So why don't we have Metrics about integer addition? If we didn't pretend they were requirements, it would make more sense -- we trust compiler writers and programmers to know what's obvious.

> The value of implementation advice seems to be in its guidance to implementers and in its documentation requirement.

Also, they let programmers know what to expect of implementations. It may be that two different implementation strategies have wildly different efficiency properties, but neither is uniformly better. With IA, we can encourage some uniformity across implementations (which are what standards are for!) and we can let programmers know what to expect.

****

From: Alan Burns
Sent: Friday, December 19, 2003 5:28 AM

> As *requirements*, the Metrics are silly. Real-time programmers use integer addition more often than they use delay statements, and they want bounded-time integer addition. So why don't we have Metrics about integer addition? If we didn't pretend they were requirements, it would make more sense -- we trust compiler writers and programmers to know what's obvious.

Yes this sums up some of the discussions that occurred at IRTAW during Ada95 process. Some people felt that metrics were key; others felt that they could never give the full story and that discussions with the complier vendor would always be needed to get the implementation model and parameters defined (if detailed timing analysis was needed). So one of the points of the metrics became to focus implementers on potential inefficiencies. Most metrics identify some parameter that we would like to see be as small as possible.

In discussions on Ravenscar, the IRTAW decided not to give a whole load of new metrics; rather it decided to add discussion on metrics etc in the Guide (being produced by HRG) on Ravenscar.

Returning to Timing Events. The point of the metric or IA is to make it clear that is would not be acceptable for an implementation to fire a timing event 10 years/months/days/minutes/seconds after the time specified. A secondary point is to find out what the maximum lateness is. In practice no new metric is needed as the same mechanism/accuracy for delay statements will be used for timing events.

So do I stay with the metric, or have IA such as 'The implementation shall fire any timing event as soon as possible after the time specified in its registration.'?

****

From: Tucker Taft
Sent: Friday, December 19, 2003 7:53 AM

I would recommend you follow the lead of other sections of the real-time systems annex (e.g. D.8) and use a mixture of Implementation Advice for largely non-quantitative issues and Metrics for largely quantitative issues.

****

From: Robert A. Duff
Sent: Friday, December 19, 2003 9:37 AM

Alan said:

> Returning to Timing Events. The point of the metric or IA is to make it clear that is would not be acceptable for an implementation to fire a timing event 10 years/months/days/mintues/seconds after the time specified.

Well, I've implemented delay statements in both Ada 83 (before metrics) and Ada 95. I didn't actually *need* to be told that "10 seconds late is too late". I figured that out on my own.

Sometimes it seems like people think that compiler vendors are liable to go around deliberately sabotaging their own products if the Standard doesn't prevent it! ;-)

> So do I stay with the metric, or have IA such as 'The implementation shall fire any timing event as soon as possible after the time specified in its registration.'?

Well, I guess WG9 wants a metric, so the expedient thing is to put in a metric. It's not your job to keep Bob Duff happy. ;-)

****

From: Ben Brosgol
Sent: Friday, December 19, 2003 10:50 AM

> Sometimes it seems like people think that compiler vendors are liable to
> go around deliberately sabotaging their own products if the Standard
> doesn't prevent it! ;-)

Wasn't there an early Ada 83 compiler that had not yet implemented preemption so its implementation of the delay statement was to suspend the task forever? It satisfied the semantics of the RM ("suspends

further execution ... for at least the duration specified ...")

****

From: Robert Dewar
Sent: Friday, December 19, 2003 3:27 PM

I believe this is urban legend. At one ARG meeting, one Robert Dewar noted that such an implementation would be compliant. Someone commented that perhaps that was a desirable implementation for nuclear missile control programs :-) :-)

****

From: Jean-Pierre Rosen
Sent: Saturday, December 20, 2003 5:30 AM

As far as I remember, this was the first version of the Meridian compiler. I remember a big argument at that time, and that Meridian eventually redid its scheduling.

****

From: Dan Eilers

Sent: Friday, December 19, 2003 11:16 AM

> So do I stay with the metric, or have IA such as 'The implementation shall fire any timing event as soon as possible after the time specified in its registration.'?

My 2 cents worth is to go very sparingly on documentation of timing issues in normative parts of the RM. Compilers are often developed using instruction-level simulators, or prototype development boards, that are substantially different from the end-user's target hardware with respect to clock speeds, cache sizes and speeds, main memory sizes and speeds, availability of hardware timers, contention from other processors, load on the system, etc.

So the implementer can address qualitative issues such as whether the runtime system supports task preemption, but not necessarily the number of microseconds or even clock cycles that something will take.

In some cases, it would be better to provide constants and/or attributes that can be queried by the software, e.g., system tick, so that the answers provided are more likely to be correct than documentation which can easily be out-of-date, or created for a somewhat different target environment, or mistakenly calculated, or completely missing.

****

From: Robert Dewar
Sent: Friday, December 19, 2003 11:46 AM

burns@cs.york.ac.uk wrote:

> Returning to Timing Events. The point of the metric or IA is to make it clear that is would not be acceptable for an implementation to fire a timing event 10 years/months/days/minutes/seconds after the time specified. A secondary point is to find out what the maximum lateness is. In practice no new metric is needed as the same mechanism/accuracy for delay statements will be used for timing events.

This seems misguided to me. An implementation is free to generate code that takes ten years to do each addition, and we don't worry ourselves about that.

The proper way to approach this is to have implementation advice suggesting what sorts of implementations are appropriate (e.g. abort is expected to be "immediate" not to wait for a synchronization point.

Note one point that the current metrics are inconsistent. Some call for guaranteed lower bounds (something that is typically impossible to give in most real life real time environments, since there are far too many layers to analyze when running on top of operating systems, and the hardware is too hard to analyze in any case, e.g. cache and TLB effects).

Other metrics ask essentially for the output of a test program. That's really silly, it simply mandates that a benchmark be written, and its results published with no idea of exactly how they were measured. It would be better to spend the effort on generating standard benchmark programs!

But if nonsense metric requirements make people happy fine, put them in, they are relatively harmless, since mostly they get ignored.

And no, we don't feel bad about ignoring them, since they are documentation requirements, they are completely undefined in my view anyway, and when it comes to documentation, we are oriented to providing what our customers find useful, the RM is really somewhat irrelevant.

We have seen neither customer demand in the metrics area, nor competitive pressure ("buy XYZ Ada compiler, unlike GNAT it has proper metrics :-)

****

From: Robert I. Eachus
Sent: Friday, December 19, 2003 10:59 PM

>So do I stay with the metric, or have IA such as 'The implementation shall fire any timing event as soon as possible after the time specified in its registration.'?

I always figure that it was one of my comments on the draft RM in 1982 that resulted in the current wording. I noticed that the then current text was incorrect. It didn't allow for the possibility that a higher priority task was executing when a delay expired. Jean thought my suggested wording was ugly, and came up with wording that looked innocuous enough. But if I had known the amount of FUD it would generate, I would still have been screaming about it.

There were words elsewhere on which tasks were eligible to run at any point in time. Jean thought that was sufficient to guarantee that a task would be scheduled immediately on the expiration of a delay, unless there was at least one higher priority task running. Instead, that wording resulted in a lot of bogus tests to check that the delay was never rounded down--and a lot of FUD like the idea that a task whose delay expired need not ever be scheduled to run. (True, but only if there was always at least one equal or higher priority task eligible to run.)

****

From: Alan Burns
Sent: Saturday, December 27, 2003 8:04 AM

There were rounds of emails on this topic but no clear conclusion on whether there should be a Metric or IA. I have no strong view either. The wording I sent used a metric - so Randy could you put the new wording into a new version of the AI (only wording should be changed) and either stay with the metric or change to AI if you feel that would be better. I'm on vacation for the next 3 weeks, so please excuse silence if there is another round of discussions.

****