Accessing Delay Queues

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

Emails: {burns,andy}.cs.york.ac.uk
Tel: +44 (0) 1904 432779
Fax: +44 (0) 1904 432767

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 scheme 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:

- runs at the right time,
- runs at the high priority, and
- 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[1].

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

[1] 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.

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 parameterless 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 cancelling 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:

- The number of task is reduced and the cost of context switching to minder tasks is eliminated.
- 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;
    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 allows 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. *IEEE Copmuter*, 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.