
Article:
Burns, A. and Wellings, A.J. (2016) The Deadline Floor Protocol and Ada. ACM Ada Letters. pp. 29-34. ISSN: 1094-3641

https://doi.org/10.1145/2971571.2971575

The Deadline Floor Protocol and Ada

Alan Burns and Andy Wellings
University of York, UK.
emails: alan.burns,andy.wellings@york.ac.uk

Abstract

This short paper summarises the current status of the proposal to incorporate the Deadline Floor Protocol into the Ada Real-Time Annex. A Draft Ada Issue is given to help focus the discussion at the workshop.

1 Introduction

At the 2013 IRTAW Workshop it was accepted that the Deadline Floor Protocol (DFP) has many advantaged over the Stack Resource Protocol (SRP), and that it should be incorporated into a future version of the language, and that ideally the support for SRP should be deprecated.

This short position paper summarises the current status of proposed language changes that would be needed to make this happen. The context is single processor systems. Furthermore, we do not consider any implications for the Ravenscar profile. The goal is to provide enough background for the workshop to develop an AI for the necessary changes to the Real-Time Annex.

2 The Deadline Floor Protocol

The DFP has all the key properties of SRP [3, 2]; specifically, causing at most a single blocking effect from any task with a longer relative deadline, which leads to the same worst-case blocking in both protocols. In an EDF-scheduled system, the DFP is structurally equivalent to Immediate Priority Ceiling Protocol (IPCP) in a system scheduled under fixed priorities.

Under the DFP, every resource has a relative deadline equal to the shortest relative deadline of any task that uses it. The relative deadline of a resource is called its *deadline floor*, making clear the symmetry with the *priority ceiling* defined for the resources in any priority ceiling protocol.

The key idea of the DFP is that the absolute deadline of a task could be temporarily shortened while accessing a resource. Given a task with absolute deadline d that accesses a resource with deadline floor D^F at time t, the absolute deadline of the task is (potentially) reduced according to $d := min(d, t + D^F)$ while holding the resource.

The action of the protocol on a single processor results in a single block per task, deadlock free execution and works for nested resource usage. Whilst a task accesses a resource its deadline is reduced so that no newly released task can preempt it and then access the resource. See [4, 5] for details and proof of the key properties. This is equivalent to the use of a priority ceiling; again the only tasks that can preempt a task executing within a protected object are tasks that are guaranteed not to use that object (unless there is a program error, which can be caught at run-time).

3 Required Language Simplifications and Modifications

To embed the rules for the DFP within Ada, the following summarises the issues must be addressed:

- All tasks must have a relative deadline assigned via an aspect or a routine defined in a library package.
- Protected objects must have also a relative deadline (floor) assigned via an aspect.
- Default relative deadline values must be defined for tasks and protected objects (and their types).

- Rules for EDF scheduling must be extended to include a new locking policy: `Floor_Locking`.
- Rules for EDF scheduling need simplifying to remove the 'across priorities' feature of the current definition.
- For completeness (and parity with priority ceilings) means of modifying the relative deadline attribute of tasks and protected objects should be defined.

We discuss how these issues are addressed below.

3.1 Relative Deadlines

Currently, only absolute deadlines are defined by the RM, and are coupled to the specification of EDF scheduling. Whilst deadlines are key to EDF scheduling, they have a wider purpose; deadlines are relevant to all forms of real-time scheduling. Moreover, programs that wish to catch and respond to missed deadlines need to be able to manipulate deadlines directly.

There are, at least, two ways of introducing relative deadlines:

Change the existing library package structure

The 2005 version of Ada introduced EDF scheduling and the subtype `Deadline`. Unfortunately, we feel, it only introduced this, as we noted above, for the support of EDF scheduling. We feel that *deadline* and *relative deadline* are fundamental concepts in real-time and deadline-aware programming [6]. We therefore propose that the whole package `Ada.Dispatching.EDF` be renamed, repositioned and extended to support relative as well as absolute deadlines. The new package could be as follows.

```
with Ada.Real_Time;
with Ada.Task_Identification;
use Ada;
package Ada.Deadlines is
  subtype Deadline is Real_Time.Time;
  subtype Relative_Deadline is Real_Time.Time_Span;
  Default_Deadline : constant Deadline :=  Real_Time.Time_Last;
  Default_Relative_Deadline : constant Relative_Deadline :=
              Real_Time.Time_Span_Last;
  procedure Set_Deadline(D : in Deadline;
            T : in Task_Identification.Task_ID := Task_Identification.Current_Task);
  function Get_Deadline(T : in Task_Identification.Task_ID :=
           Task_Identification.Current_Task) return Deadline;
  procedure Set_Relative_Deadline(R : in Relative_Deadline;
            T : in Task_Identification.Task_ID := Task_Identification.Current_Task);
  function Get_Relative_Deadline(T : in Task_Identification.Task_ID := Task_Identification.Current_Task)
                                 return Relative_Deadline;
   procedure Delay_Until_And_Set_Deadline( Delay_Until_Time : in Real_Time.Time;
            Deadline_Offset : in Real_Time.Time_Span :=  Get_Relative_Deadline);
end Ada.Deadlines;
```

Key changes are:

- Change of name and library position.
- Introduction of a type for relative deadline and a default value.
- Set and Get routines added for relative deadlines.
- A default relative deadline provided for `Delay_Until_And_Set_Deadline`.

All tasks will have a deadline and a relative deadline; default values being used if the program does not specify specific values.

As with priority, where a task has a base and an active priority, a task will also have a base (absolute) deadline and an active (absolute) deadline – see definition of the locking policy below. A call of `Get_Deadline` returns the base deadline of the task.

The existing aspect Relative_Deadline should be redefined to take an expression of type Relative_Deadline. Note, although the same name is used here, this is the same situation with subtype Priority and aspect/pragms Priority. However, the definition of the aspect Relative_Deadline should really be moved from D.2.6.

Keep the current library structure and add new package

The alternative is to just provide a new child package of Ada.Dispatching.EDF.

```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF.Dynamic_Relative_Deadlines is
  subtype Relative_Deadline is Real_Time.Time_Span;
  Default_Relative_Deadline : constant Relative_Deadline :=
              Real_Time.Time_Span_Last;
   procedure Set_Relative_Deadline (D : in Relative_Deadline;
                  T : in Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task);
  function Get_Relative_Deadline (T : Ada.Task_Identification.Task_Id :=
                   Ada.Task_Identification.Current_Task) return Relative_Deadline;
end Ada.Dispatching.EDF.Dynamic_Relative_Deadlines;
```

3.2 Setting and Changing the Deadline Floor of a Protected Object

A new aspect is required to assign deadline floors to protected objects:

```
protected Object with Deadline_Floor => Ada.Real_Time.Milliseconds(24) is ...
```

To dynamically change the deadline floor of a protected object a new Deadline attribute should be provided. This new attribute would behave very much like the Priority attribute:

```
protected body PO is
  procedure Change_Relative_Deadline (D: in Real_Time.Time_Span) is
  begin
     ... -- PO'Deadline has old value here
    PO'Deadline := D;
     ... -- PO'Deadline has new value here
  end Change_Relative_Deadline; -- relative deadline is changed here
...
end PO;
```

Detecting floor violations could require a new check.

3.3 New Locking Policy

The proposed new dispatching policy identifier (EDF_With_Deadline_Floor) is intended to be used in the Task_Dispatching_Policy and the Priority_Specific_Dispatching pragmas:

```
pragma Task_Dispatching_Policy (EDF_With_Deadline_Floor);
pragma Priority_Specific_Dispatching (EDF_With_Deadline_Floor, first_priority, last_priority);
```

Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Currently EDF dispatching is supported via the policy EDF_Across_Priorities. A range of priorities is needed to account for the different priority ceilings needed for the protected objects. The tasks themselves only execute at the base priority of this range when they are not executing within a protected action. All ready queues are ordered by the (absolute) deadline of the ready tasks.

To prevent confusion, and to emphasis the fact that with the new protocol only a single priority is needed for all EDF dispatched tasks (regardless of the number of protected objects they use), we propose a new dispatching policy. And to accommodate hierarchical dispatching (see Section 4) we define the new policy as EDF_Within_Priorities. We will not attempt to give a full definition appropriate for the ARM[1].

[1]For example, consideration would need to be given to whether deadline inheritance should occur during a rendezvous and task activation, and whether entry queues can be deadline ordered.

With EDF_Within_Priorities, all tasks with the same priority compete for the processor using the rules for EDF dispatching. The ready queue is ordered by *active* deadline. A collection of EDF dispatched tasks and the set of protected objects they use/share will all have the same priority (and ceiling priority). But they will have different relative deadlines (and deadline floors).

A task that has not been given an explicit deadline or relative deadline will get the default values of Default_Deadline (equal to : Real_Time.Time_Last) and Default_Relative_Deadline (equal to Real_Time.Time_Span_Last). The default value for the deadline floor of any protected object is 0 (actually TimeSpan_Zero). This will have the effect of making all protected actions non-preemptive (as does the default priority ceiling).

Another issue is related to the Ceiling_Locking policy. Locking policies are applied to the whole partition using the Locking_Policy pragma, so it could be argued that the identifier name Ceiling_Locking is inappropriate since its use for a partition implies that a protocol different from the Ceiling Locking is going to be used in EDF_With_Deadline_Floor priority ranges. On the other hand, it could be considered that for EDF scheduling the DFP is the equivalent to the Ceiling Locking concept and then the identifier would be appropriate. More relevant than the identifier name are the deep modifications that would be required in the definition of the ceiling locking policy (RM D.3). Currently this policy is only defined in terms of priorities and, with the incorporation of the DFP, it would be necessary to define it also in terms of deadlines, i.e.

- Whenever a task is executing outside a protected action, its active deadline is equal to its base deadline.
- When a task executes a protected action its active deadline will be reduced to (if it is currently greater than) 'now' plus the deadline floor of the corresponding protected object.
- When a task completes a protected action its active deadline returns to the value it had on entry.
- When a task calls a protected operation, a check is made that there is no task currently executing within the corresponding protected object; Program_Error is raised if this check fails.

With this definition of a new locking policy, the definition of Ceiling_Locking can return to its pre-2005 wording.

Note the semantics requires a check on non-concurrent access to the protected object. It is not sufficient to check that the relative deadline of the task is not less than the deadline floor of the object. This points to a difference with Ceiling_Locking where a comparison based on priorities is sufficient. To implement the check on inappropriate usage over the corresponding protected object requires only a simple 'occupied' flag to be checked and modified. Usefully, if there is an attempt to gain access to an occupied protected object then the task 'at fault' is forced, on a single processor, to be the second task that is attempting to gain access, and it will therefore be this task that has the exception raised. The correct task will be unaffected.

Interestingly, a simple check on non-concurrent access would also be sufficient for the priority ceiling case. And again the exception is bound to be raised in the task 'at fault'. Of course, checking concurrent access, rather than correct priority/ceiling values will only catch an actual error rather than a potential one. Inappropriate ceiling values will be caught on first usage, inappropriate concurrent access may be very difficult to create during testing. Although not a sufficient test, it might be advisable to also include in the definition of Floor_Locking a static check on the relative deadlines of user tasks and the deadline floors of the used protected objects.

To ensure that locking protocols work correctly, the programmer must give the correct values for deadline floors and ceiling priorities. A run-time check prevents concurrent access, but a compiler-based check cannot be undertaken and hence the use of the correct values can only be asserted by code inspection or static analysis.

4 Hierarchical and Mixed Scheduling

One of the advantages of the new EDF_Within_Priorities policy is that it unifies Ada's use of priority as the primary dispatching policy. It is no longer necessary to reserve a range of priorities for a single EDF domain. If we ignore the non-preemptive policy, we now have a clear means of supporting mixed scheduling in a hierarchical manner:

- At all times, the task at the head of the highest priority non-empty ready queue is the one chosen to be executed.
- Each ready queue has its own discipline to determine which task is at its head.

The disciplines supported are: FIFO, Round Robin (RR) and now EDF; i.e. `FIFO_Within_Priorities`, `Round_Robin_Within_Priorities` and now `EDF_Within_Priorities`.

4.1 Interaction between EDF_With_Deadline_Floor and FIFO_Within_Priorities

The first situation to consider is when the EDF range is at a higher priority level than the FIFO_Within_Priorities range. In such a situation a FIFO task could use protected objects in the EDF range to interact with EDF tasks. The FIFO task would inherit both the priority ceiling and the deadline floor of the protected object. Deadline floors of protected objects in the EDF priority band are assigned according to the deadlines of the EDF tasks that access them. To avoid a deadline floor violation when they are used by a FIFO task, it is enough to assume that the relative deadline of the FIFO tasks is infinite (Ada.Real_Time.Time_Span_Last). Consequently, a FIFO task accessing an EDF protected object would inherit its deadline floor and could only be preempted by EDF tasks with shorter relative deadlines. This behavior is expected since, by definition, those tasks with shorter relative deadlines are not going to access the protected object. In the opposite situation, when the EDF range is below the FIFO_Within_Priorities range, only the basic ceiling locking policy rules go into action: the EDF task that accesses a protected object in the FIFO range inherits the priority ceiling of the protected object and, consequently, can only be preempted by tasks with higher priorities.

4.2 Interaction between EDF_With_Deadline_Floor and Round_Robin_Within_Priorities tasks

It is exactly the same situation as for the FIFO_Within_Priorities policy since the round robin tasks behave like FIFO tasks while executing a protected action (the quantum does not expire until the protected action finishes).

4.3 Interaction between two EDF_With_Deadline_Floor tasks in different priority ranges

Although the utility of having more than one EDF priority range could be arguable, the language must be complete and should take into account that possibility. It is the user's responsibility to avoid deadline floor violations when a task from the lower priority range uses a protected object in the higher priority range. The deadline floors assigned to the protected objects should consider the deadlines of all the tasks that access them in both priority ranges.

5 Summary of 2013 Discussion

Several issues were raised at IRTAW 17 discussion and these are summarised below [7], most of these have already addressed.

- The impact of release jitter on the correctness of the protocol. Michael Gonzalez Harbour explained that care had to be taken when tasks could be subject to release jitter as this could result in the delayed execution of a shorter deadline task that then could preempt a longer deadline task while it was active in the protected object. It was, therefore, necessary to use the values of $Deadline - Jitter$ for each task rather than its simple deadline. Failure to do this would invalidate the protocol, and mutual exclusion would not be guaranteed by the protocol itself. Hence, for safety it is also necessary to provide a mutex lock to control protected object access. It was noted, that a similar problem occurs with jitter and the priority ceiling protocol. However, there more priority inversion results instead of the breaking of mutual exclusion. It was also noted that it was possible to optimize the lock so that it was a single bit that indicate that the protected object is occupied. Any attempt to access an occupied protected object would result in an exception being raised.
- The meaning of an inherited deadline. In a real-time system there are usually consequences that must be managed if a task misses its deadline. With the deadline floor protocol, a task may inherit a deadline, which will be shorter than its application-defined deadline. The workshop discussed the consequences of a task missing its inherited deadline. It was agreed that inherited deadlines were required to control scheduling and missing them had no repercussions for the application tasks. For example, the default floor for a protected object is $Time\_Span\_First$, and hence it is quite possible that an absolute deadline computed using this floor value is missed. Consequently, the workshop recommended that, similar to priorities, that there should be a notion of *base* and *active* deadline. The programmer would have no visibility of the active deadline of a task. Any application-level deadline detection mechanisms involves its base rather than its active deadline.

- Protected objects shared between EDF-scheduled and priority-scheduled tasks. In order to fit into the Ada framework for scheduling mixed systems, it is necessary to allow some protected objects to have *both* a priority ceiling and a deadline floor. The rules are simple, if the ceiling of the protected object is a FIFO-within priority level, the task's active deadline is not updated while executing within the protected object (i.e. there is no need to have a deadline floor). If the ceiling priority is an EDF-within priority level, the task's active deadline is updated (i.e. it does need a floor). Nested protected object across levels require further consideration.

- Dynamic changes to the base deadline. It was noted that asynchronous changes to the base deadline of a task does not result in the recalculation of any active deadline associated with the task. Also a new optional check could be specified when using `Delay_Until_And_Set_Deadline` to ensure that the new deadline is longer than or equal to now plus the relative deadline of the user tasks (as set by the pragma `Relative_Deadline`).

- Deadlines and other inheritance points in Ada. For completeness, the workshop agreed that in principle a server task should run with an active deadline which is the shortest of its own deadline and the deadline of the calling tasks during a rendezvous between two tasks. Similarly, deadline inheritance should occur during task activation.

6 Conclusions

This paper has summarised the current status of the proposal to support the Deadline Floor protocol in the next revision of Ada. We feel that the proposal is now mature enough to warrant the production and submission of an AI.

Acknowledgements

This paper contains no new material but draws together the material currently available on the DFP. We acknowledge all those that have engaged in the development of the protocol [1, 5, 6].

References

[1] M. Aldea, A. Burns, M. Gutirrez, and M. G. Harbour. Incorporating the deadline floor protocol in Ada. *ACM SIGAda Ada Letters – Proc. of IRTAW 16*, XXXIII(2):49–58, 2013.

[2] T. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.

[3] T. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[4] A. Burns. A Deadline-Floor Inheritance Protocol for EDF Scheduled Real-Time Systems with Resource Sharing. Technical Report YCS-2012-476, Department of Computer Science, University of York, UK, 2012.

[5] A. Burns, M. Gutirrez, M. Aldea, and M. G. Harbour. A Deadline-Floor Inheritance Protocol for EDF Scheduled Embedded Real-Time Systems with Resource Sharing. *IEEE Transaction on Computers, 64(5), pp 1241-1253*, 2015.

[6] A. Burns and A. J. Wellings. Deadline-Aware Programming and Scheduling. *Proceedings Reliable Software Technologies, Ada-Europe*, LNCS 8454, pp 107–118, 2014.

[7] A. Wellings. Session summary: Locking protocols. *ACM SIGAda Ada Letters, Proc. of IRTAW 16*, XXXIII(2):123–125, 2013.