Document downloaded from:

http://hdl.handle.net/10251/121375

This paper must be cited as:

Real Sáez, JV.; Sáez Barona, S.; Crespo, A. (2018). Ravenscar Support for Time-Triggered Scheduling. ACM SIGAda Ada Letters. 38(1):41-54.
https://doi.org/10.1145/3241950.3241957

The final publication is available at

http://doi.org/10.1145/3241950.3241957

Copyright Association for Computing Machinery

Additional Information

Ravenscar Support for Time-Triggered Scheduling*

Jorge Real†, Sergio Sáez‡, Alfons Crespo†
† Instituto de Automática e Informática Industrial
‡ Instituto Tecnológico de Informática
Universitat Politècnica de València
Camino de vera, s/n, 46022 Valencia, Spain

E-mail: {jorge,ssaez,alfons}@disca.upv.es

Abstract

This position paper follows from a previous proposal to integrate a time-triggered scheduler in a priority-based, preemptive scheduler such as that supported by Ada's task dispatching policy FIFO_Within_Priorities *. The resulting combined scheduling carries the advantages of both time-triggered and priority-based scheduling, and helps mitigating their drawbacks.*

The paper presents a system model for the time-triggered subsystem that extends the original proposal, and describes a Ravenscar implementation of the scheduler at the run-time system level, in the form of a new package Ada.Dispatching.TTS. *Multiple programming patterns can be implemented on top of this scheduler. With respect to the previously proposed full-Ada implementation, only patterns that implied the use of asynchronous transfer of control have been excluded. On the other hand, the extension of the original model enables new patterns, not supported in our previous proposal, using the new types of continuation and optional slots.*

We hold that bringing the time-triggered paradigm to Ravenscar is both feasible and convenient for the High-Integrity and Embedded application domains.

Keywords: Real-Time Systems. Time-Triggered Scheduling. Priority-Based Scheduling. Ravenscar Profile. High-Integrity Systems. Embedded Systems

1 Introduction

Time-Triggered (TT) and Priority-Based (PB) scheduling are two approaches often taken in exclusion. In previous publications [9, 8], we have proposed a model, and its corresponding Ada implementation, that combines both scheduling techniques in such manner that a real-time application can be implemented by a combination of TT and PB tasks. The TT subset of tasks is executed at the highest priority of a preemptive PB scheduler, hence granting time slots for the TT workload over the PB subset of tasks. This separation allows one to keep the advantages of TT scheduling (determinism, minimum jitter) for a selected subset of TT tasks, and to retain the properties of PB scheduling (separation of logic/timing concerns) for tasks that do not require the extreme predictability of a TT schedule. This approach was well received at a previous International Real-Time Ada Workshop [6].

*This work has been partly supported by the Spanish Government's project M2C2 (TIN2014-56158-C4-1-P-AR) and the European Commission's projects ENABLE-S3 and AQUAS (ECSEL-JU, Contracts 692455 and 737475).

In this paper we want to make this approach usable in the context of the Ravenscar Profile [2], hence making it adoptable in the High-Integrity and Embedded Systems domains.

At the time of writing this paper, we have an ongoing submission [7] that presents the system model and a set of patterns that can be supported by the TT scheduler in combination with the PB runtime of Ravenscar. We will only briefly describe these aspects here. Quantitative results are also given in [7], showing a very tight and predictable jitter for TT tasks; but we will not repeat them here. Instead, we will develop the discussion about design and implementation details that are given in [7].

2 System Model: the Time-Triggered Plan

The description of our system model is limited to the aspects related with the TT workload. The PB subset of tasks is scheduled at lower priorities than the TT level, hence causing no application interference on TT tasks.

Since we are after a Ravenscar implementation of the model, the priority-based subset will be scheduled using a fixed-priority scheme such as Rate Monotonic or Deadline Monotonic [5, 4]; but nothing in our model precludes the use of other algorithms for the PB tasks, such as EDF [5], or even a combination of schedulers using different priority bands.

A TT plan is a cyclic sequence of actions to be executed at particular points in time. The plan is described by an ordered list of *time slots*, each of its own *slot duration*. If a slot starts at time t, its lifetime goes from t to t+*Slot Duration*. There are no gaps between slots: each slot starts just at the end of the previous slot in the plan.

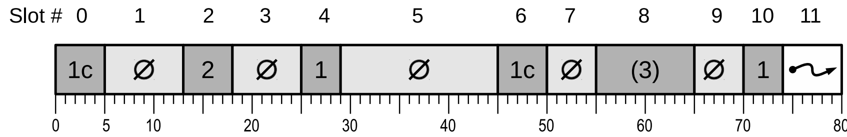

Figure 1. A 12-slot time-triggered plan. Slots marked 2, 4 and 10 are regular slots for works 1 and 2, as indicated; slots 0 and 6 are continuation slots for work 1; slot 8 is an optional slot for work 3; and slot11 is a mode change slot. The rest are empty slots.

Figure 1 shows a 12-slot example plan, with slots numbered from 0 to 11. Apart from its duration, each slot is characterised by the actions to take during the slot lifetime. Each slot has a mark indicating its type (digits and other symbols whose meaning will be described in a moment). Using the time scale in milliseconds at the bottom of the plan, it can be seen that it has a duration of 80 ms, slot 0 has a duration of 5 ms, slot 11 takes 6 ms from time 74 ms to 80 ms, etc.

Our system model defines five possible types of slots in a plan:

- A ***regular slot*** is a time slot reserved for the execution of a TT task (or *work*). It is denoted by a *regular Work_Id*, a strictly positive integer that identifies the particular work to execute during the slot. In Figure 1, slots 2, 4 and 10 are regular slots corresponding to works 1 or 2 as indicated. The underlying TT scheduler will make the work start to execute as soon as feasible after the start time of the slot.

  The duration of a regular slot must be sufficient, by design, to accommodate the worst-case execution time of the work it serves. If a work overruns its regular slot then the scheduler will resort to raising a Program_Error, since that would violate the schedulability assumptions of TTS. If, on the contrary, a work completes before the end of the slot duration, then the rest of the slot remains available for PB tasks. A TT task must always be ready to use its allocated regular slots in the plan. Failing this, the scheduler will raise Program_Error as well.

- An ***optional slot*** is like a regular slot except that it can be omitted, hence the associated work does not need to be ready when the slot arrives. If the work is ready at the start of the slot, then the optional slot has the same semantics as a regular slot, including overrun control. But if it is not ready, then this is not

considered an error and the slot duration is made available for PB tasks. In Figure 1, slot 8 is an optional slot for work 3, indicated with parentheses.

Optional slots are useful for tasks that may or may not require to use their allocated slot, such as a communication task when it has nothing to say; or a sporadic task whose activation event has not occurred.

- A ***continuation slot*** is a special kind of regular slot. In Figure 1, slots 0 and 6 are continuation slots. They are marked with a regular Work_Id plus a letter 'c', indicating continuation.

  What is special about them is that there is no overrun control: if a work does not complete by the end of a continuation slot, then it it is held now and resumed when its next slots arrives. This hold/resume mechanism is indeed to be taken very carefully, specially with regard to its interaction with protected actions; but it is doable under certain restrictions as we will show later.

  There may be consecutive continuation slots for a given work. Overrun will only be checked when the plan reaches a regular slot for this work. We refer to the last, non-continuation slot of a series, as a *terminal slot*. In Figure 1, regular slots 4 and 10 are terminal for work 1, given that they are preceded by continuation slots 0 and 6, respectively.

  Continuation slots are useful to split a large time-triggered task into smaller pieces in a way that is essentially transparent to the task code.

The following two types of slots correspond do not have an associated TT task to execute.

- An ***empty slot*** defines an interval during which no TT activity is planned. This is useful for inserting gaps in the TT plan to make the CPU available to PB tasks. In Figure 1, slots 1, 3, 5, 7 and 9 are empty slots.
- A ***mode-change slot*** defines the times in the plan where it is possible to substitute the current plan with a new one. There is no TT task to execute during a mode change slot. This type of slot allows the designer to determine the points where the system can admit a mode change. At the start of a mode-change slot, the TT scheduler will check whether there is a pending mode-change request to process. If there is one, then the new plan will start executing at the end of the mode-change slot. The change will be immediate if the mode-change slot duration is defined to be zero. In Figure 1), slot 11 is a mode change slot, indicated with a curved arrow.

For comparison with the TT plan model we defined in previous papers [9, 8], the model we have just defined introduces the new types of continuation and optional slots. The former are motivated by feedback received from participants at the 18$^{\text{th}}$ International Real-Time Ada Workshop [6].

Finally, we assume a single processor platform, although the model is applicable to multiprocessor systems provided that they are fully partitioned, i.e., there is only one plan per processor and tasks statically fixed to CPUs (no migration).

3 Time-Triggered Task Patterns

There are a number of possible patterns of TT tasks that can be supported by the model sketched in Section 2. The paper [7] describes eight of them in detail and derives requirements for the TT scheduler to support them. The following list is a summary of what is proposed there.

Simple TT Task Pattern A simple TT task. There is one regular slot in the plan for each full execution of this type of work. The pattern implements an infinite loop with a call to the scheduler to wait for the next regular slot marked with the task's Work_Id (Wait_For_Activation), and then the sequence of statements implementing the work actions. At the beginning of the slot, the scheduler releases the task from its call to Wait_For_Activation. At the end of the slot, the scheduler checks for overrun.

A simple TT task may have its own local state, since TT tasks are supported by regular Ravenscar tasks. It can also share data with other simple TT tasks, because it executes in mutual exclusion with other simple TT tasks (there are no overlapping slots). If the task needs to share data with preemptable PB tasks (or sliced TT tasks, as we'll see later), then it needs to do it via protected objects. In such case, the TT task may experience blocking that must be taken into account when deciding the slot duration.

Initial-Final Pattern (I-F) This pattern is for TT tasks that can be subdivided in two parts, both with strict jitter requirements. It is easily obtained by sequential composition of two simple TT patterns: an infinite loop split in two parts, first the initial and then the final, both using the same Work_Id when calling Wait_For_Activation. Note that the slots for the initial and final parts need not have the same duration. Overrun is checked for both parts.

Initial-Mandatory-Final Patterns (I-M-F and I-{M}-F) This pattern (I-M-F, for short) uses three consecutive regular slots to perform a logically related sequence of TT actions. This scheme is typical of embedded control systems, with initial, mandatory and final parts performing data acquisition, processing and output, respectively. I-M-F is obtained by concatenation of three simple patterns in the loop body: three calls to Wait_For_Activation preceding the statements of the initial, mandatory and final parts. Overrun is checked for each and every part of the task.

This pattern can be generalised to a form I-{M}-F, where there are one or more slots dedicated to execute parts of the mandatory section, each part with overrun control. This requires explicit slicing of the mandatory part in the task's code.

Sliced TT Task Pattern This pattern allows us to distribute the execution of a long running TT task across several slots. A sliced TT task uses one or more continuation slots, ending with a regular slot (the *terminal slot* of the sliced sequence). Slicing occurs transparently, i.e., it does not require explicit calls to Wait_For_Activation. If the task has not called Wait_For_Activation at the end of the slot, then it is held by the TT scheduler and will be resumed at the start of its next continuation or regular (terminal) slot. The pattern does not differ apparently from a simple TT task. The difference resides in the type of slots reserved for the task in the plan.

Holding and resuming are not possible unless the TT scheduler is implemented at the runtime level, with access to task control blocks and scheduler queues. Hence our proposal to include the TT scheduler at the Ada library level, as a new package Ada.Dispatching.TTS.

Since a sliced task can be held asynchronously, data sharing with other tasks can only be via protected objects for a sliced task, at a ceiling priority that effectively disables the scheduler, i.e., at the highest interrupt priority. Only at this ceiling can it be guaranteed that the sliced task is not held in the middle of a protected action. As a consequence, it is specially important that protected actions involving a sliced TT task are as short as possible. If the protected action cannot be so short, then there are still alternatives. One is to design the plan so that all continuation slots are followed by empty slots of sufficient duration to absorb the potential blocking time. And if this is not possible, because the data exchange required is too large, then the application code may resort to using multiple-buffering techniques to reduce the need for mutual exclusion to the minimum.

A final consideration with sliced TT tasks is that mode change slots should not be inserted in the middle of sliced sequences, because track of the sequence may be lost when switching from one plan to another.

Initial-Mandatory_Sliced-Final Patterns (I-Ms-F and IMs-F) The I-Ms-F pattern is a variant of the I-M-F pattern where the mandatory part is sliced. The IMs-F pattern (note the missing dash between the 'I' and 'M' parts) is a slight modification of the former that allows the mandatory part to start executing immediately after the initial part, without waiting for the next slot in the plan. Both patterns have the same representation in the plan, taking one regular slot for the initial part (so that it is subject to overrun control), then several continuation slots for the mandatory part, and one regular slot for the final part.

The IMs-F pattern requires specific support from the TT scheduler. Since IMs-F allows the mandatory sliced part to start as soon as the initial part is done, during the first regular slot, we are effectively transforming the semantics of this particular regular slot into that of a continuation slot. The scheduler must therefore be informed of the termination of the initial part so that, if the initial part is not done by the end of the slot, then there is an overrun; but if it has completed, then the slicing regime has started and the hold/resume mechanism has to apply to the already started sliced mandatory part. The procedure to inform the scheduler of the start of the sliced mandatory part is called Continue_Sliced.

TT Patterns with Non-TT Parts The patterns with non-TT parts are possible thanks to the scheduler functionality represented by procedure Leave_TT_Level. With this function, a TT task running at the TT level

informs the scheduler that it wants to continue executing at a priority below the TT level, possibly in competition with other, higher-priority PB tasks. This is useful to execute parts that are not subject to strict jitter requirements and that may be difficult to integrate in the TT plan.

As an example, consider a control task with jitter-sensitive initial and final parts. An intermediate part tries to improve a quick control action calculated in the initial part by means of an optimisation algorithm that takes disparate execution times depending on changing environment conditions. If this middle part had to be included in the TT plan, then it would require sufficient slots for its worst-case execution. But since we can execute the optimisation part below the TT level, then we require slots in the plan, hence keeping it simpler. At the end of the middle part, the task goes back to the TT level via Wait_For_Activation to execute the final part with minimal jitter. Schedulability analysis must guarantee that the optimisation phase will meet the start of the final slot.

The Leave_TT_Level mechanism requires changing the priority of the TT task at runtime, which, at first sight, appears to be in contradiction with the Ravenscar model of fixed priorities. However, a Ravenscar runtime has to actually support a limited form of dynamic priorities, because it is needed to implement Ceiling_Locking. Support to Leave_TT_Level can be implemented as well in Ravenscar, using this already available runtime facility. The problem can be seen as if the task had a base priority in the PB level, at which it runs its PB phase, and an active priority at the TT level when it runs in a TT slot. The mechanism does not need to change the priority of a task other than the running task, as for Ceiling_Locking case. And the changes between base (PB) and active (TT) priorities occur only as a result of calls to Leave_TT_Level and Wait_For_Activation executed by the same task that is affected by the priority changes, as is the case for protected actions.

Leave_TT_Level can also be used to compose other patterns. For example P-F (for Priority-Based and Final), a periodic PB task with one TT phase, to be executed during a regular slot in the plan. This slot can be used for mutually exclusive communication or data exchange with other tasks, or for accessing a shared resource in general, such as in a slot-based communication protocol.

Another case is the P-[F] pattern (a PB part followed by an optional final part), which uses a non-TT part and an optional slot. This pattern fits a periodic, PB task, that may or may not use a slot at the end of an iteration. At the TT level, the task requires just one optional slot per iteration. At the PB level, the task executes as any other periodic or sporadic task. So this is a variant of P-F for tasks that could decide to skip the TT slot without causing Program_Error for that.

4 Design and Implementation Details

This section presents the design of the TT scheduler and some implementation details. The complete implementation of version 0.1 of the TT scheduler can be found in [10].

As it follows from the model described in Section 2 and from the services required by continuation slots and non-TT parts as shown in Section 3, the proposed scheduler is provided as a runtime extension in the form of a new package Ada.Dispatching.TTS, mimicking the position of the EDF package in Ada.Dispatching.

4.1 Scheduler Specification

For reduced memory footprint, we propose the package Ada.Dispatching.TTS to be generic, with one generic parameter specifying the number of regular Work_Ids in the plan. This will allow the implementation to adjust the number of data structures needed to keep track of all works in the plan to just the number required by the application.

Listing 1 shows the visible part of the package specification, plus assignment of two deferred constants in the private part. It gives definitions of types related to Work_Ids, TT slots and the type Time_Triggered_Plan, representing the plan as an array of slots. An access type allows to efficiently pass a plan as a parameter in procedure calls. The specification then declares procedures Set_Plan, needed to establish a new TT plan or to replace the current plan (i.e., a mode change at the TT level), and procedures Wait_For_Activation, Continue_Sliced and Leave_TT_Level, already mentioned before. In this version, Wait_For_Activation includes an out parameter to inform the caller about the start time of the slot, according to the plan. We have used this value to measure the release jitter of TT tasks.

Listing 1. Specification of the proposed generic package Ada.Dispatching.TTS

```
with Ada.Real_Time;

generic

   Number_Of_Work_IDs : Positive;

package Ada.Dispatching.TTS is

   -- Work identifier types
   type Any_Work_Id is new Integer range Integer'First .. Number_Of_Work_IDs;
   subtype Special_Work_Id is Any_Work_Id range Any_Work_Id'First .. 0;
   subtype Regular_Work_Id is Any_Work_Id range 1 .. Any_Work_Id'Last;

   -- Special IDs
   Empty_Slot       : constant Special_Work_Id;
   Mode_Change_Slot : constant Special_Work_Id;

   -- A time slot in the TT plan
   type Time_Slot is record
      Slot_Duration   : Ada.Real_Time.Time_Span;
      Work_Id         : Any_Work_Id;
      Is_Continuation : Boolean := False;
      Is_Optional     : Boolean := False;
   end record;

   -- Types representing/accessing TT plans
   type Time_Triggered_Plan        is array (Natural range <>) of Time_Slot;
   type Time_Triggered_Plan_Access is access all Time_Triggered_Plan;

   -- Set new TT plan to start at the end of the next mode change slot
   procedure Set_Plan (TTP : Time_Triggered_Plan_Access);

   -- TT works use this procedure to wait for their next assigned slot.
   -- The When_Was_Released result informs caller of slot starting time
   procedure Wait_For_Activation (Work_Id : Regular_Work_Id;
                                  When_Was_Released : out Ada.Real_Time.Time);

   -- TT works use this procedure to inform that the critical part
   -- of the current slot has completed
   procedure Continue_Sliced;

   -- TT works use this procedure to inform the TT scheduler that
   -- there is no more work to do at TT priority level
   procedure Leave_TT_Level (Work_Id : Regular_Work_Id);

private

   Empty_Slot       : constant Special_Work_Id := 0;
   Mode_Change_Slot : constant Special_Work_Id := -1;
   ...
end Ada.Dispatching.TTS;
```

The time-triggered plan is represented by an array of slot descriptors, each with one field to indicate the slot duration and another three fields to characterise the slot as follows:

- Work_Id - This field contains either a positive value identifying a TT task, or an indication of *empty slot* or *mode change slot*, via the corresponding constants.
- Is_Continuation - When this boolean is True, the slot is marked as a *continuation slot*.
- Is_Optional - A boolean that marks the slot as an *optional slot*.

The slot descriptor contains relevant information for the scheduler. Another important source of information is the current status of TT tasks. This part is however internal to the scheduler and hence a part of the package body.

4.2 Scheduler body

The scheduler maintains a data structure, the work control block (WCB), that stores the work status to determine the actions to be taken during a slot switch. The status of a TT work is determined by the following boolean fields of a WCB record:

- Has_Completed - Indicates whether a single-slot work or a sliced work does not require more time at TT level:. This is set to True when the TT task calls Wait_For_Activation or Leave_TT_Level.
- Is_Waiting - Indicates whether the work task is waiting for a new slot or not. This flag is set to True when the work calls Wait_For_Activation.
- Is_Sliced - When this flag is True, it means that this work is currently running sliced, hence it may need to be held/resumed. This flag is set to True when the work enters a continuation slot or when it invokes Continue_Sliced. It is set to False when the work is at the start of a terminal slot.

Listing 2 shows the complete structure of WCB, and the declaration of an array of the required size, depending on the value given to Number_Of_Work_IDs in the instantiation of Ada.Dispatching.TTS. The other fields in the Work_Control_Block record are Release_Point, the suspension object where the work waits for its next activation; Work_Thread_Id, a low level identifier of the runtime-level thread behind the task supporting the work; and Last_Release, the time of the last release of the work, the out parameter of Wait_For_Activation for jitter instrumentation.

Listing 2. First part of the body for package Ada.Dispatching.TTS, including the declaration of the Work Control Block structure

```
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
with Ada.Real_Time;                use Ada.Real_Time;
with System.BB.Threads.Queues;
-- Other context clauses omitted ...

package body Ada.Dispatching.TTS is
   use System.BB.Threads;

   -- Run time TT work info
   type Work_Control_Block is record
      Release_Point  : Suspension_Object;
      Has_Completed  : Boolean   := True;
      Is_Waiting     : Boolean   := False;
      Is_Sliced      : Boolean   := False;
      Work_Thread_Id : Thread_Id := Null_Thread_Id;
      Last_Release   : Time      := Time_Last;
   end record;

   -- Array of Work_Control_Blocks
   WCB : array (Regular_Work_Id) of Work_Control_Block;
   ...
```

The TT scheduler is the handler of a recurrent timing event set to trigger at the start of each slot in the plan. Hence it executes at the highest interrupt priority (ARM D.15 12/2 [3]). The protected object containing the handler for this timing event, protects scheduler's internal variables such as accesses to the current and the next plan, current and next slot indexes for traversing the plan, and the start time of the next slot.

Based on the slot information and WCB, the scheduler decides the actions to take during a slot switch, i.e., the time when a slot starts and the previous slot finishes. Table 1 describes some of these actions. The top part of Table 1 lists two cases of actions to take at the end of a regular or continuation slot. These actions can be *Hold task*, to hold the running TT task when it exhausts a regular slot and must continue sliced in future slots; and *raise Program_Error* when overrun is detected, i.e., the task has not completed and it is not running sliced. To support Hold, our implementation uses runtime operations (available from System.BB.Threads and its child unit

Queues to suspend the low-level thread behind the TT task and extract it from the ready queue. This is why a sliced part can only use protected objects with priority ceiling at the highest interrupt priority, as mentioned in Section 3 with regard to the Sliced TT Pattern, because *Hold* should not happen while the TT task is executing a protected action.

The bottom part of Table 1 lists scheduler actions related to the immediately starting slot. The actions that are common to most cases in this table (denoted *CA*) are to mark the work as sliced when it enters a continuation slot (the WCB inherits this slot property), and to set the timing event handler to the end of this starting slot, at the Next_Slot_Release time. Transferring the Is_Continuation property of the slot to the Is_Sliced attribute of the work effectively propagates the *sliced* condition of the TT task until the terminal slot, for which Is_Continuation is False.

Work status		**Actions at END of slot**
Has_Completed	Is_Sliced	
False	True	Hold task
False	False	Raise `Program_Error`

Work status			Next Slot	**Actions at START of a REGULAR slot**
Has_Completed	Is_Sliced	Is_Waiting	Is_Optional	
True	True			Common Actions (CA)*
True	False	True		Release Task + CA*
True	False	False	True	CA*
True	False	False	False	Raise `Program_Error`
False	True			Resume Task + CA*

*Common Actions ≡ Work. Is_Sliced := Slot . Is_Continuation ; Set_Handler(Next_Slot_Release , Handler)

Table 1. Some actions taken by the scheduler at a slot switch. Actions with regard to the exhausted slot (*Actions at END of slot*) are shown in the top part, Actions related to the immediately starting slot (*Actions at START of a REGULAR slot*) are listed at the bottom part. The actions to take depend on work status (in the WCB) and the type of slot.

The scheduler must also perform some actions upon calls to its public services, Wait_For_Activation, Continue_Sliced and Leave_TT_Level. These may affect the work status and the priority of the underlying threads of caller TT tasks. It is the task itself who changes its own work status and priority, if needed, while running a scheduler protected operation. Table 2 summarises these update operations. For example, when a TT task invokes Leave_TT_Level, its work status is marked as completed and its priority demoted to the task's base priority – so the base priority of the task implementing the TT pattern must be determined according to the required priority level when it runs in the PB level.

Invoked procedure	**Changes to work status**			**Task prio set to**
	Has_Completed	Is_Sliced	Is_Waiting	
`Wait_For_Activation`	True		True	`Priority'Last`
`Continue_Sliced`		True		
`Leave_TT_Level`	True			`Task'Base_Priority`

Table 2. How and when the work status and TT task priority are modified by the scheduler.

To end this description, Table 3 shows the actions taken by the scheduler at the start of special slots (*empty* and *mode change* slots). When te slot is an empty slot, the scheduler just reprograms itself to take action at the

end of the slot. In case of a mode change slot, if there is a new plan pending to switch to, due to a previous call to Set_Plan, then the mode change is enforced, that is, the current plan is changed with the next and indexes in the new plan are reset to start it from the beginning. This is achieved by a call to the scheduler's internal protected subprogram Change_Plan. If there is no new plan set, then the mode change is treated as an empty slot.

Next Slot Type	∃ Next Plan	**Actions at START of a SPECIAL slot**
Empty		Set_Handler(Next_Slot_Release , Handler)
Mode Change	False	Set_Handler(Next_Slot_Release , Handler)
Mode Change	True	Change_Plan (Next_Slot_Release)

Table 3. Some actions to be taken by the scheduler at the start of special slots.

5 Conclusions

This paper has presented the results of transforming a full-Ada architecture for combined TT-PB scheduling [9, 8] to make it Ravenscar-compatible. Our aim was to bring this scheduling strategy closer to a more appropriate programming model for High-Integrity and Embedded systems. Compared to our previous full-Ada implementation, this Ravenscar implementation does not support patterns that imply cancellation of TT works. This pattern was supported in the full-Ada version using asynchronous transfer of control in the TT patterns, something that can not be ported to Ravenscar. But the Ravenscar-compliant model we have proposed here supports the rest of patterns we had previously proposed and the new continuation slots (via hold/resume) and non-TT slots (via priority demotion), thus improving expressiveness and making room for more patterns. We have also introduced the concept of optional slot, whereby no-show situations, where a TT task decides not to use a slot in the plan, are now tolerable.

In addition, we have made Ada.Dispatching.TTS a generic package, where the number of regular work identifiers is a generic parameter. This allows us to keep the size of data structures to the minimum necessary for the number of TT tasks to be scheduled. The experimental results are encouraging, even better than those obtained in full-Ada with a much faster processor. No doubt, the simplicity of the Ravenscar runtime has to do with these results. Besides, Jitter is very predictable in the STMF32F4 Discovery board where we have tested our implementation, which enables optimisations that keep release jitter for TT tasks in the order of a few microseconds [7].

References

[1] M. Aldea and M. González-Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. *Reliable Software Technologies - Ada Europe 2001, Lecture Notes in Computer Science*, 2043:305–316, 2001.

[2] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high-integrity systems. Technical Report YCS-2017-348, University of York, June 2017.

[3] ISO/IEC-JTC1-SC22-WG9. *Ada Reference Manual ISO/IEC 8652:2012(E)*. URL: `http://www.ada-europe.org/manuals/LRM-2012.pdf`, 2012.

[4] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, 1982.

[5] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[6] J. Real and P. Rogers. Session Summary: Experience. *Ada Letters*, 36(1):101–102, June 2016.

[7] J. Real, S. Sáez, and A. Crespo. Combined Scheduling of Time-Triggered and Priority-Based Task Sets in Ravenscar. *Submitted for publication - International Conference on Reliable Software Technologies - Ada-Europe 2018.*

[8] J. Real, S. Sáez, and A. Crespo. Combined Scheduling of Time-Triggeed Plans and Priority Scheduled Task Sets. *Ada Letters*, 36(1):68–76, June 2016.

[9] J. Real, S. Sáez, and A. Crespo. Combining Time-Triggered Plans with Priority Scheduled Task Sets. In M. Bertogna and L. M. Pinho, editors, *Reliable Software Technologies – Ada-Europe 2016*, volume 9695 of *Lecture Notes in Computer Science*. Springer, June 2016.

[10] S. Sáez and J. Real. TTS Ravenscar runtime. https://doi.org/10.5281/zenodo.1168723, February 2018.