In support of extending the Ravenscar profile

Jorge Garrido Beatriz Lacruz Juan Zamorano

Juan A. de la Puente

Abstract

This paper discusses different approaches for implementing an EEPROM memory driver which is part of the UPMSat2 satellite on-board computer software. The Ravenscar profile restrictions are to be observed in order to ensure the analysability of the system, and therefore the approaches are evaluated against the profile. Results of this evaluation as well as considerations on a possible extension of the Ravenscar profile with respect protected entries are presented.

1 Introduction

The Ravenscar profile [1, D.13] allows a restricted subset Ada tasking in certain kinds of critical systems where predictability, efficiency, and static analysis are required. It has been successfully used in many real-time applications, and has enabled concurrency to be used in many situations where full Ada tasking is not allowed.

However, there are some cases in which using the profile is difficult and can lead to complex programs and abstraction inversion. At the last IRTAW meeting Rogers, Ruiz, and Gingold [3] proposed relaxing some of the restrictions while still keeping the advantages of predictability and efficiency. The rationale behind their proposal is that the current specifications in the Real-Time Annex make it possible to implement some additional tasking features in a predictable way. In particular, they showed how an implementation of an extended profile with a bounded number of protected entries and bounded entry queues can be implemented in the Ravenscar GNAt runtime while keeping the worst-case execution time (WCET) and memory space bounded, with an acceptable amount of overhead in execution time.

The purpose of this position paper is to support this proposal by showing a real example in which the use of the Ravenscar profile leads to unnecessary efficiency, whereas an extended profile allowing multiple entries in a protected object enables a much better implementation. The example is a subset of an EEPROM driver which is part of the UPMSat2 on-board software [2].

2 EEPROM memory in UPMSat2

EEPROM (Electrically Erasable Programmable Read-Only Memory) is a type of non-volatile memory that can be erased by means of electrical pulses. The On-Board Computer (OBC) of the UPMSat2 satellite has two EEPROM chips, each one with a size of 1 MB, making up 2 MB. The first

half of the memory space is used to store the executable OBC code, and the rest is used to store parameters and data. This part of the memory is divided into blocks of two different sizes. There are 16 blocks of 512 B, which are used to store configuration parameters, and 4032 blocks of 128 B, which are used to store telemetry and telecommand messages.

Due to the technical specifications of the EEPROM, a minimum wait interval of 15 ms must be kept after a write operation before starting a new read or write operation, in order to prevent the written data to be erased. In order to obtain the required bandwidth, the EEPROM block mode, allowing up to 128 consecutive word write accesses, is used. It must be noted that block writes must be atomic.

3 Ravenscar EEPROM drivers

3.1 Basic design

The EEPROM driver is to be accessed by different software tasks, and therefore a basic approach is to use a protected object to ensure mutual exclusive access:

```
protected EEPROM
   with Priority => System.Interrupt_Priority'Last;
is
   procedure Write (B : in  Memory_Block);
   -- includes an active wait loop

   procedure Read  (B : out Memory_Block);
end EEPROM;
```

However, there are two main issues with this design:

- As stated before, a delay of 15 ms has to be enforced at the end of the `Write` operation. Since blocking operations are not allowed in the body of protected operations, `delay` may not be used to this purpose. The immediate solution is thus to implement the delay by means of an active wait loop.
- Write operations must be executed in an atomic way. In order to avoid interrupts from other devices to interfere wit them, the priority of the protected object must be set to `System.Priority'Last`, as above shown. This is enough to inhibit all interrupts on a mono-processor platform, as is the case in the UPMSat2 OBC.

In consequence, a call to the `Write` procedure may block any other task for up to 15 ms, while the procedure is executing the active wait loop. This is clearly undesirable, and therefore a better solution has to be found.

3.2 A dissociated design

A better solution, which is currently used in UPMSat2, is to dissociate the active wait from the `Write` operation. Two protected objects are used, an external object which implements the external interface and the active wait loop, and an internal object which carries out the write operation.

```
protected EEPROM  -- internal object
   with Priority => System.Interrupt_Priority'Last;
is
   procedure Write (B : in  Memory_Block);
```

```
   -- no active wait

   procedure Read  (B : out Memory_Block);

end EEPROM;

protected EEPROM_Interface -- external object
   with Priority => ...    -- set to priority ceiling of callers
is
   procedure Write (B : in  Memory_Block);
   -- calls EEPROM.Write and then executes and active wait loop

   -- the Read procedure dos not need to be wrapped
   -- as it does not have to be atomic
end EEPROM_Interface;
```

This approach provides the benefit of not executing the active wait at the highest priority. However, as stated before, the `EEPROM_Interface` object is called from many tasks in the system, thus inheriting a high executing priority under the ceiling priority protocol. As a result, although the priority at which the active wait loop is executed has been reduced, most tasks in the system are still experiencing long blocking times.

4 Extended Ravenscar drivers

4.1 Synchronized task approach

A third approach would be to do the wait after block writings outside any protected object, thus being able to use a delay statement in a low priority server task rather than an active wait. This approach would highly reduce the interference caused by the writing delay. However, it requires a more sophisticated synchronization mechanism for writing and reading operations. This is usually achieved in Ada by means of protected entries.

However, Ravenscar restrictions do not allow any possible implementation with a reasonable level of complexity. Neither a solution with a single entry queueing the calls nor a set of entries discriminating operations and tasks are acceptable under the profile.

It could be claimed that a solution with a single entry for activating the server task and procedure operations for queueing requests may be achievable even with the Ravenscar profile restrictions. But several issues arise at this point, the most relevant one being the complexity of implementing the queueing protocol. Furthermore, this implementation would be redundant, as the runtime has already implemented a queueing mechanism for entry calls.

4.2 A better solution

In fact, the most desirable approach for the EEPROM driver implementation would be to have a protected object with two entry queues, one for `Write` operations and another one for `Read` operations. As shown by Rogers et al. [3], this would not compromise the schedulability analysis, or add a significant timing overhead.

The code scheme for this approach is listed below.

```
1  package EEPROM is
2     type Memory_Block is ... ;
3     procedure Read  (B : out Memory_Block );
4     procedure Write (B : in Memory_Block );
```

```
5  end EEPROM;
```

```
1  with System;
2  with Ada.Real_Time;
3  use type Ada.Real_Time.Time_Span;
4
5  package body EEPROM is
6
7     protected EEPROM
8         with Priority => System.Interrupt_Priority'Last
9     is
10        procedure Write (B : in Memory_Block;
11                         Busy_Period_Start : out Ada.Real_Time.Time);
12    end EEPROM;
13
14    task Server
15       with Priority => Server_Task_Priority; -- A low priority task
16
17    protected EEPROM_Interface
18       with Priority => EEPROM_Ceiling
19    is
20        entry Write (B : in Memory_Block);
21        entry Read  (B : out Memory_Block);
22        entry Start_Server_Task (Busy_Period_Start : out Ada.Real_Time.Time);
23        procedure Finish_Wait;
24    private
25        Busy : Boolean := False;
26        Busy_Period_Start : Ada.Real_Time.Time;
27    end EEPROM_Interface;
28
29    procedure Read (B : out Memory_Block) is
30    begin
31        EEPROM_Interface.Read (B);
32    end Read;
33
34    procedure Write (B : in Memory_Block) is
35    begin
36        EEPROM_Interface.Write (B);
37    end Write;
38
39    protected body EEPROM_Interface is
40
41        entry Write (B : in Memory_Block) when not Busy is
42        begin
43            Busy := True;
44            EEPROM.Write (B, Busy_Period_Start);
45        end Write;
46
47        entry Read (B : out Memory_Block) when not Busy is
48        begin
49            -- Read a block from EEPROM
50            null;
51        end Read;
52
53        entry Start_Server_Task (Busy_Period_Start : out Ada.Real_Time.Time)
54              when Busy is
55        begin
56            Busy_Period_Start := Busy_Period_Start;
```

```
57          end Start_Server_Task;
58
59          procedure Finish_Wait is
60          begin
61              Busy := False;
62          end Finish_Wait;
63      end EEPROM_Interface;
64
65      task body Server is
66         Busy_Period_Start : Ada.Real_Time.Time;
67      begin
68         loop
69            EEPROM_Interface.Start_Server_Task (Busy_Period_Start);
70            delay until Busy_Period_Start + Ada.Real_Time.Milliseconds (15);
71            EEPROM_Interface.Finish_Wait;
72          end loop;
73      end Server;
74
75      protected body EEPROM is
76
77          procedure Write (B : in Memory_Block;
78                           Busy_Period_Start : out Ada.Real_Time.Time) is
79          begin
80             -- perform an EEPROM block writing
81             Busy_Period_Start := Ada.Real_Time.Clock;
82          end Write;
83
84      end EEPROM;
85
86  end EEPROM;
```

5 Conclusions

The Ravenscar profile imposes some restrictions on the Ada tasking model that have been previously recognised as too strict. Previous work has shown that it is possible to relax some of the restrictions without incurring significant penalties in terms of performance and predictability.

Here we have presented a realistic case study where such restrictions impose an inefficient implementation. We have shown how some of the proposed extensions to the Ravenscar profile may improve the quality and performance of the solution. The case study can be extended with other hardware drivers in the UPMSat2 system that have similar problems.

References

[1] ARM12. *ISO/IEC 8652:2012(E): Information Technology — Programming Languages — Ada*, 2012.

[2] J. Garrido, J. Zamorano, J. A. de la Puente, A. Alonso, and E. Salazar. Ada, the programming language of choice for the UPMSat-2 satellite. In *Data Systems in Aerospace — DASIA 2015*. Eurospace, 2015.

[3] P. Rogers, J. Ruiz, and T. Gingold. Toward extensions to the Ravenscar profile. *Ada Letters*, 35(1):32–37, April 2015.