Execution-Time Clocks and Ravenscar Kernels [*]

Juan Antonio de la Puente
Departamento de Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid
jpuente@dit.upm.es

Juan Zamorano
Departamento de Arquitectura y Teconología de Computadores
Universidad Politécnica de Madrid
jzamora@fi.upm.es

Abstract

The kind of high-integrity real-time systems to which the Ada Ravenscar profile is targeted often require overrun detection for enhanced reliability in the time domain. Execution-time clocks and timers have been proposed to fulfill this need, but only programming patterns based on asynchronous transfer of control, and thus incompatible with the profile, have been provided up to now. In this paper an assessment of the compatibility of the proposed execution-time mechanisms with the Ravenscar profile is made, and some programming patterns for using them are proposed. The patterns are intended to provide basic overrun detection and handling capabilities to high-integrity real-time systems.

1. Introduction

Hard-real time systems have strict timing requirements, which can be guaranteed by construction, using an ad-hoc cyclic executive [7, 23] or by analysis on a multitasking executive, using response time analysis techniques [12]. Multitasking has clear advantages over cyclic executives, because of its inherent flexibility and ease of maintenance [18], and is becoming the preferred approach even in such fields as embedded spacecraft systems [21] where the cyclic executive method has traditionally been chosen.

Response time analysis techniques require a computational model with specific characteristics in order to ensure a predictable behaviour in the time domain. An example of such a model is the Ada 95 Ravenscar profile [8, 9, 11], which was defined at the 8th IRTAW and further refined at subsequent IRTAW meetings [6, 22, 10]. The profile has been included in the ISO report on the use of Ada in high integrity systems [17], and the Ada Rapporteur Group (ARG) has accepted it as a proposed addition to the next revision of the language.

Ensuring the required real-time behaviour usually relies on an accurate knowledge of the worst-case computation times (WCET) of all the real-time tasks. Although some good techniques for computing WCET are available [20], there is still a large degree of uncertainty, especially when modern processors with cache memories and segmentation are used. Pessimistic WCET estimations lead to an underutilization of resources, and thus tight estimates are usually sought. The risk with tight WCET estimates is, on the other hand, to be optimistic, and then occasionally get an actual execution time which is larger than the estimated value. This situation is called an *overrun*, and may give rise to a generalised miss of deadlines by tasks by a domino effect. Run-time monitoring techniques are often used to detect when a task surpasses its budgeted execution time and take corrective actions before other tasks are affected by an overrun.

Monitoring of the actual execution times is routinely done in cyclic executives by means of a simple mechanism: every time the timer interrupt arrives, if there is a task running then some task has exceeded its budgeted execution time, and a recovery action is invoked. The recovery action often takes the form of replacing the faulty task, or a group of related tasks, by an emergency task that either provides a degraded functionality or performs a safe stop. However, this simple kind of overrun detection cannot be used in multitasking systems such as the ones built with Ada tasking and the Real-Time annex.

González-Harbour and others [16] proposed to add ex-

[*]The work described in this paper has been partially funded by the Spanish Ministry of Science and Technology (MCYT), project TIC2002-04123.

ecution clocks to Ada in order to provide this functionality. A similar mechanism has been included in the POSIX real-time services. A proposal to incorporate the POSIX execution clock mechanisms in Ada was presented and discussed at IRTAW'10 [2], and was demonstrated in MarteOS [3]. Based on this work a proposal was discussed at the last IRTAW and presented to the ARG (AI95-00307). The original proposal contemplated two possible implementations, one based on a library package, and the other one including full integration in the language, with the definition of a new time type [19]. Although the integrated solution is better from the programmer's point of view, the package solution was preferred by the ARG because it does not require any changes to the language or to existing compilers.

The new Ada execution time clocks can be used to detect overruns and to implement several recovery strategies [16] with Ada tasking. However, their use with the Ravenscar profile and their implementation in Ravenscar kernels has not been investigated. The rest of this paper discusses the compatibility of execution time clocks with the Ravenscar profile, the overrun detection and recovery mechanisms that can possibly be used with the profile, and the possible ways of including execution time clocks in Ravenscar kernels. Only the package solution is considered as it is the one preferred by the ARG.

2. Compatibility with the Ravenscar profile

The first question to be solved is whether the execution clock mechanisms themselves are compatible with the Ravenscar profile. For this purpose let us look first at the proposed Ada. Real_Time. Execution_Time package specification (appendix A). Its main elements are:

- The CPU_Time type. This data type is not intended to be a "time type" as defined in the ALRM (9.6.6.) [1], i.e. it cannot be used in delay until statements and other timing constructs.
- A Clock function that provides an execution-time clock for each task.
- A protected Timer type definition from which execution-time timers can be declared on a per-task basis.

  Timers can be armed and disarmed, and tested for expiration. An entry (Time_Exceeded) allows a task to wait until the timer expires.

Checking these definitions against the Ravenscar profile restrictions, it is clear that none of them are violated. In particular, the definition of the Timer protected type has only one entry, which can be easily implemented with a simple barrier. It can thus be concluded that the use of the Ada.Real_Time.Execution_Time package is compatible with the Ravenscar profile.

3. Overrun detection and handling in Ravenscar programs

3.1. Overrun detection schemes

Most of the overrun detection and recovery schemes proposed by González-Harbour and his colleagues [16] cannot be used under the Ravenscar profile restrictions, as they make use of such mechanisms as asynchronous transfer of control (ATC), abort statements, or dynamic priorities, which are not allowed by the profile. This means that alternative overrun detection and handling techniques have to be used. We propose here two such schemes which can be used to detect overruns in real-time tasks. Overrun handling strategies are then discussed in section 3.2.

Handled overrun detection

The first overrun detection scheme is the same as the *handled* scheme which was previously proposed by González-Harbour *et al*. The code for a periodic task is given in program 1.

```
-- Program 1: Handled overrun detection

with Ada.Real_Time,
     Ada.Real_Time.Execution_Time,
     Ada.Task_Identification;
use Ada;
use type Ada.Real_Time.Time;
use type Ada.Real_Time.Execution_Time.Time;
...
task body Periodic_Handled is
   Next_Start : Real_Time.Time:=Real_Time.Clock;
   WCET : constant Duration:=1.0E-3;
   Period : constant Duration:=1.0E-2;
   T : Ada.Real_Time.Execution_Time.Time;
begin
   loop
      T := Ada.Real_Time.Execution_Time.Clock;
      Do_useful_work;
      if Ada.Real_Time.Execution_Time.Clock - T > WCET then
         -- budgeted execution time exceeded
         Handle_the_error;
      end if;
      Next_Start := Next_Start + To_Time_Span(Period);
      delay until Next_Start;
   end loop;
end Periodic_Handled;
...
```

This scheme lets a task check if it has executed for longer that its budgeted WCET once it has completed its periodic activity. Therefore, if there is an overrun, it is only detected after the harm has been done, and some other tasks may have missed their deadlines by that time. A more effective scheme, which enables fast detection of overrun conditions, is given in the next section.

Supervised overrun detection

The second scheme uses an execution-time timer and a supervisor task to detect possible overruns in a real-time task. The supervisor task is blocked on the Timer_Expired entry of the timer, which is armed when the real-time task starts its activity and disarmed when it is completed. Under normal conditions the timer is disarmed before its expiration, but if the task activity takes longer to execute than its budgeted WCET, the timer entry barrier is opened and the supervisor task gets ready to run. The supervisor is assigned a very high priority so that it preempts the execution of the faulty task and can thus do some corrective action without waiting to the completion of the real-time task activity. The code for a periodic task is given in program 2.

```
-- Program 2: Supervised overrun detection

with Ada.Real_Time,
     Ada.Real_Time.Execution_Time,
     Ada.Task_Identification;
use Ada;
use type Ada.Real_Time.Time;
use type Ada.Real_Time.Execution_Time.Time;
...
Periodic_Id : aliased Task_Identification.Task_Id :=
   Periodic_Supervised'Identity;
The_Timer : Ada_Real_Time_Execution_Time.Timer
   (Periodic_Id'Access);

task body Periodic_Supervised is
    Next_Start : Real_Time.Time:=Real_Time.Clock;
    WCET : constant Duration:=1.0E−3;
    Period : constant Duration:=1.0E−2;
    T : Ada.Real_Time.Execution_Time.Time;
begin
    loop
        The_Timer.Arm(Real_Time.To_Time_Span(WCET));
        Do_useful_work;
        The_Timer.Disarm;
        Next_Start := Next_Start + To_Time_Span(Period);
        delay until Next_Start;
    end loop;
end Periodic_Supervised;

task Supervisor is
    pragma Priority(System.Priority'Last);
end Supervisor;

task body Supervisor is
begin
    loop
        The_Timer.Timer_Expired;
        Handle_the_error;
    end loop;
end Supervisor;
...
```

This scheme is similar to the *lowered* and *stopped* schemes proposed in [16] in that the overrun is detected by a separate supervisor task. It differs from these schemes in that it does not use any programming constructs which are forbidden by the Ravenscar profile.

3.2. Handling overruns

When an overrun is detected, some action has to be performed in order to preserve the integrity of the system. While the recovery strategy to be used is application dependent, there are some general principles that can be stated for a wide variety of systems.

The effects of an overrun may involve several tasks in addition to the faulty one, as having a task executing for a longer time than budgeted increases processor utilisation and may result in loss of schedulability and other tasks missing their deadlines. Of course, this may happen even if the tasks are fully independent and have no logical relationships among them. Therefore, the recovery action must be undertaken at the system level rather than at the individual task level.

A common approach in traditional systems based on a cyclic executive is to replace the faulty task, and in some cases a group of tasks related to it (its *recovery group*) by a single *recovery task* implementing a minimum functionality [23]. This method works well with cyclic executives because overruns can be detected at the end of a scheduling cycle, and thus an alternative schedule using a different set of tasks can be used in the next cycle. Extending the approach to multitasking systems is not straightforward, as the transition from a set of tasks to another one may involve transient overloads that can make the system unschedulable [4, 5]. Task group recovery schemes for multitasking systems were proposed in [14], and simpler approaches based on stopping the faulty task or lowering its priority so that it lets other tasks execute have been proposed in [16]. However, all of these schemes are based either on asynchronous transfer or control, or the ability of a supervisor task to abort other tasks. Indeed, stopping the execution of a task or a group of tasks at asynchronous instants (i.e. when an overrun is detected), requires the supervisor task to have some control over the execution of ordinary real-time tasks.

The Ravenscar profile, however, excludes all kinds of abort mechanisms, as well of asynchronous transfer of control, as they may lead to unpredictable behaviour and thus compromise the integrity of the system. This means that none of the above mentioned recovery schemes can be used with high-integrity systems using Ravenscar tasking.

The need for fault recovery and system reconfiguration in Ravenscar programs is a general issue which is not limited to overrun faults. The previous IRTAW discussed a proposal (AI95-0266) for associating protected procedures to a task, which are invoked when the task is about to terminate, either normally or abnormally [10]. The initial proposal, which includes a task group mechanism, was rejected at the workshop, but it was later reformulated to a simpler scheme. However, the proposal does not include a task termination mechanism other than abortion, and it thus does not solve the problem of starting a system reconfiguration from a supervisor task.

Therefore, there is a need to provide a system-level service for reconfiguration after an overrun has been detected. This functionality has to be provided by the run-time system, as it comes from the previous discussion that no application-level mechanism is available to this purpose under the Ravenscar profile restrictions. The reconfiguration services could be provided by a library package such as the one shown in program 3.

```
-- Program 3: Example of reconfigurationpackage

with Ada.Task_Identification; use Ada.Task_Identification;
package Ada.System_Reconfiguration is

   type Recovery_handeler is access protected procedure;

   procedure Set_Safe_Stop_Procedure (Handler: Recovery_Handler);
   -- called by application code to provide an application-level
   -- safe stop procedure

   procedure Set_Reset_Procedure (Handler: Recovery_Handler
   -- called by application code to provide an application-level reset

   -- the following procedures can called by supervisor tasks to initiate
   -- task recovery actions

   procedure Safe_Stop;

   procedure Reset;

   procedure Stop_Task (Id : in Task_Id);

end Ada.System_Reconfiguration;
```

Such a package would enable a supervisor task to immediately stop the faulty task and possibly start a replacement task (using the same pattern as proposed in [11] for mode changes and other kinds of system reconfiguration), or to perform a safe stop or even a system reset, depending on the system requirements.

4. Implementing execution-time clocks in Ravenscar kernels

Pilot implementations of execution-clocks in Marte OS [3] suggest that implementing this functionality in Ravenscar kernels will not be too complex. A pilot implementation on the Open Ravenscar Kernel [13, 15] has been made, although it has a different interface than the AI proposal.

5. Conclusions

The discussions and examples provided in the previous sections show that execution-time clocks are a indeed a useful feature for high-integrity real-time systems. The compatibility of the execution-time package which has been proposed for the next revision of the language with Ravenscar profile has been stated, and some possible uses of this mechanism for overrun detection and recovery have been proposed.

6 Acknowledgments

The authors wish to acknowledge Alejandro Alonso and and Rémi Desclodure's help with the pilot implementation of execution-time clocks on ORK.

References

[1] *Ada 95 Reference Manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995*, 1995. Available from Springer-Verlag, LNCS no. 1246.

[2] M. Aldea and M. G. Harbour. Extending Ada's real-time systems annex with the POSIX scheduling services. *Ada Letters*, XXI(1):20–26, March 2001. Proceedings of the 10th International Real-Time Ada Workshop, Las Navas, Ávila, Spain, September 200.

[3] M. Aldea and M. G. Harbour. MaRTE OS: An Ada kernel for real-time embedded applications. In A. Strohmeier and D. Craeynest, editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in LNCS, pages 305–316. Springer-Verlag, 2001.

[4] A. Alonso and J. A. de la Puente. Implementing mode changes and fault recovery for hard real-time systems in ada. In L. Boullart and J. A. de la Puente, editors, *Real-Time Programming 1992. Proceedings of the IFAC/IFIP Workshop*. Pergamon Press, 1992.

[5] A. Alonso and J. A. de la Puente. Dynamic replacement of software in hard real-time systems. In *5th Euromicro Workshop on Real-Time Systems*. IEEE Computer Society Press, 1993. ISBN 0-8186-4110-X.

[6] L. Asplund, B. Johnson, and K. Lundqvist. Session summary: The Ravenscar profile and implementation issues. *Ada Letters*, XIX(25):12–14, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[7] T. Baker and A. Shaw. The cyclic executive model and Ada. *Real-Time Systems*, 1(1), 1989.

[8] T. Baker and T. Vardanega. Session summary: Tasking profiles. *Ada-Letters*, XVII(5):5–7, 1997. Proceedings of the 8th International Ada Real-Time Workshop.

[9] A. Burns. The Ravenscar profile. *Ada Letters*, XIX(4):49–52, 1999.

[10] A. Burns and B. Brosgol. Session summary: Future of the Ada language and language changes such as the Ravenscar profile. *Ada Letters*, XXII(4), December 2002. Proceedings of the 11th International Real-Time Ada Workshop.

[11] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003.

[12] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3 edition, 2001.

[13] J. A. de la Puente, J. F. Ruiz, and J. Zamorano. An open Ravenscar real-time kernel for GNAT. In H. B. Keller and E. Ploedereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

[14] J. A. de la Puente, J. Zamorano, A. Alonso, and J. L. Fernández. Reusable executives for hard real-time systems in Ada. In J. van Katwijk, editor, *Ada: Moving Towards 2000. Proceedings of the Ada-Europe Conference*. Springer-Verlag, 1992.

[15] J. A. de la Puente, J. Zamorano, J. F. Ruiz, R. Fernández, and R. García. The design and implementation of the Open Ravenscar Kernel. *Ada Letters*, XXI(1), 2001.

[16] M. González-Harbour, M. Aldea, J. Gutiérrez, and J. C. Palencia. Implementing and using execution time clocks in Ada hard real-time applications. In L. Asplund, editor, *Reliable Software Technologies — Ada-Europe'98*, number 1411 in LNCS, pages 90–101. Springer-Verlag, 1998.

[17] ISO/IEC/JTC1/SC22/WG9. *Guide for the use of the Ada Programming Language in High Integrity Systems*, 2000. ISO/IEC TR 15942:2000.

[18] C. D. Locke. Software architectures for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[19] J. Miranda and M. G. Harbour. A proposal to integrate the POSIX execution-time clocks into Ada 95. In J.-P. Rosen and A. Strohmeier, editors, *Reliable Software Technologies — Ada-Europe 2003*, number 2655 in LNCS. Springer-Verlag, 2003.

[20] P. Puschner and A. Burns. A review of worst-case execution time analysis. *Real-Time Systems*, 18(2/3):115–128, May 2000.

[21] T. Vardanega. Development of on-board embedded real-time systems: An engineering approach. Technical Report ESA STR-260, European Space Agency, 1999. ISBN 90-9092-334-2.

[22] A. Wellings. 10th International Real-Time Ada Workshop — Session summary: Status and future of the Ravenscar profile. *Ada Letters*, XXI(1), March 2001.

[23] J. Zamorano, A. Alonso, and J. A. de la Puente. Building safety critical real-time systems with reusable cyclic executives. *Control Engineering Practice*, 5(7), July 1997.

A Execution time package specification

```
with Ada.Task_Identification;
package Ada.Real_Time.Execution_Time is

   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last : constant CPU_Time;
   CPU_Time_Unit : constant := implementation-defined-real-number;
   CPU_Tick : constant Time_Span;

   function Clock
     (T : Ada.Task_Identification.Task_ID
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;

   function "+" (Left : CPU_Time; Right : Time_Span)
     return CPU_Time;
   function "+" (Left : Time_Span; Right : CPU_Time)
     return CPU_Time;
   function "-" (Left : CPU_Time; Right : Time_Span)
     return CPU_Time;
   function "-" (Left : CPU_Time; Right : CPU_Time)
     return Time_Span;

   function "<" (Left, Right : CPU_Time) return Boolean;
   function "<=" (Left, Right : CPU_Time) return Boolean;
   function ">" (Left, Right : CPU_Time) return Boolean;
   function ">=" (Left, Right : CPU_Time) return Boolean;

   procedure Split(T : CPU_Time;
                   SC : out Seconds_Count;
                   TS : out Time_Span);

   function Time_Of (SC : Seconds_Count; TS : Time_Span)
     return CPU_Time;

   protected type Timer
     (T : access Ada.Task_Identification.Task_ID)
   is
      procedure Arm (Interval : Time_Span);
      procedure Arm (Abs_Time : CPU_Time);
      procedure Disarm;
      entry Timer_Expired;
      function Timer_Has_Expired return Boolean;
      function Time_Remaining return Time_Span;
   private
      ... -- not specified by the language
   end Timer;

end Ada.Real_Time.Execution_Time;
```