Managing Multiple Execution-Time Timers from a Single Task

Michael González Harbour and Mario Aldea Rivas

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005-Santander, SPAIN
{mgh, aldeam}@unican.es

***Abstract**: A proposal for adding execution time clocks and timers to the Ada language through a new package called* `Ada.Real_Time.Execution_time` *is being considered by the ARG. In that proposal, execution time budgets can be managed through protected objects that provide an entry to detect execution time budget overruns. With that interface, a given task can only wait for one budget overrun, thus making it impossible for a single task to manage the execution-time budgets of several other tasks. In this paper we propose an extension to the* `Execution_Time` *package that would allow a monitoring task to manage multiple budgets.*

Keywords: *Real-Time Systems, Execution time budgets, CPU time, Ada 95.*

1. Introduction[1]

In hard real-time systems it is essential to monitor the execution times of all tasks and detect situations in which the estimated worst-case execution time (WCET) is exceeded. This detection was usually available in systems scheduled with cyclic executives, because the periodic nature of its cycle allowed checking that all initiated work had been completed at each cycle. In event-driven concurrent systems the same capability should be available, and this can be accomplished with execution time clocks and timers.

Moreover, many flexible real-time scheduling algorithms require the capability to measure execution time and be able to perform scheduling actions when a certain amount of execution time has been consumed (for example, sporadic servers in fixed priority systems, or the constant bandwidth server in EDF-scheduled systems). Support for execution time clocks and timers will ease implementation of such flexible scheduling algorithms in Ada 95 and is being considered by the Ada Rapporteur Group (ARG) for inclusion in the next revision of the language standard [6].

The proposed approach lacks the ability to manage several execution-time budgets from a single task. We consider that this functionality is important to reduce the amount of tasks dedicated to monitoring activities, and to ease cooperative management. Consequently, in this paper we propose a simple extension to the `Execution_Time` package, adding a new protected object that can store a set of execution time timers, and with an operation to simultaneously wait for the expiration of any of them.

In Section 2 of the paper we show an overview of the current proposal. Section 3 describes our proposed extension; an example of its usefulness is shown in Section 4. Section 5 discusses some implementation issues. In Section 6 we propose adding a cancellation functionality to the timers defined in the current proposal. Finally, Section 7 contains some conclusions.

2. Overview of the Execution-Time package

This is a summary of the most relevant parts of the proposed `Ada.Real_Time.Execution_Time` package in [6].

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

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

  -- Arithmetic and relations functions for
  -- managing CPU_Time and Time_Span omitted

  -- Split and Time_Of conversion operations
  -- omitted

   protected type Timer
     (T : access Ada.Task_Identification.Task_ID)
   is
      procedure Arm (Interval : Time_Span);
```

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC 2002-04123-C03 and by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project)

```
      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;

   Timer_Error : exception;
   -- may be raised by Timer_Expired,
   -- Timer_Has_Expired, and Time_Remaining

   Timer_Resource_Error : exception;
   -- may be raised on the declaration of a
   -- Timer or calls to either Arm

private
   ... --  not specified by the language
end Ada.Real_Time.Execution_Time;
```

The `CPU_Time` type represents execution time of a given task as measured approximately from its activation. The `Clock` function returns the execution time of the given task. Execution time budgets are managed through objects of the protected type `Timer`, which represent software timers that are able to detect an execution time overrun, specified either in a relative or absolute way, for a given task.

A timer can be in one of two states: armed or disarmed. It is created in association with a given task in the disarmed state, and it can be armed later through one of the `Arm` protected operations. When armed, the timer starts counting execution time until it expires. Expiration can be detected asynchronously through the `Timer_Has_Expired` operation. It can also be detected synchronously, by waiting upon the `Timer_Expired` entry call. Among other usage schemes, this entry call may be used in an asynchronous transfer of control to abort the execution of some piece of code that overrun its budget. References [3] and [6] include some usage examples for this kind of timer.

In addition to the described proposal, some time ago an implementation of execution-time clocks and timers that was integrated into the language was presented [5]. In that proposal, `CPU_Time` was an Ada time type and could be used in delay statements. The ARG has chosen the package approach presented in this section because it is easier to implement as it does not require changes to the compiler.

One major difference between the two proposals is that in the proposal integrated into the language it was possible for a given task to manage the budgets of several other tasks, by using a select statement with several delay alternatives:

```
loop
  select
    accept Set_Task_Budget  ...
  or
    delay until Task_1_Budget;
    -- lower task 1 priority
  or
    delay until Task_2_Budget;
    -- lower task 3 priority
  or
    delay  until Task_3_Budget;
    -- lower task 3 priority
  end select;
  -- recalculate budgets
end loop;
```

This functionality is not possible with the original package proposal, because a given task cannot wait upon several `Timer_Expired` entry calls simultaneously. In the next section we show an extension to this package that would make this functionality possible.

3. Extension of the Execution-Time Package

To allow a given task to manage multiple budgets it is necessary that there is an operation that allows the task to simultaneously wait for several timer expirations. This wait semantics is better described through a protected entry, that allows the calling task to use it in conjunction with the different select statements (i.e, conditional entry call, timed call, or asynchronous transfer of control).

Consequently, we propose adding the following type, protected type, and exception to the `Ada.Real_Time.Execution_Time` package:

```
   type Timer_Ref is access all Timer;

   protected type Set_Of_Timers (Max : Integer)
   is
      procedure Add    (T : in Timer_Ref);
      procedure Delete (T : in Timer_Ref);
      entry Timer_Expired
        (Cancelled : out Boolean;
         Reason : out Integer;
         T : out Timer_Ref);
      procedure Cancel (Reason : in Integer);
   private
      ... -- not defined by the language
   end Set_Of_Timers;

   Max_Timers_Error : exception;
   -- Raised by Add if total number exceeds Max
```

The protected type `Set_Of_Timers` internally stores a set of references to protected objects of the `Timer` type. It has a discriminant, `Max`, that sets the maximum number of different timers that may be added to the set. The set is managed through the `Add` and `Delete` operations. Adding a timer already belonging to the set or deleting a non included one have no effects, as is usual in set data structures. `Add` may fail raising `Max_Timers_Error` if an attempt is made to exceed the maximum number `Max`. A given timer may be added to different sets.

The protected object has an internal cancellation state and associated reason. Initially the cancellation state is not set; it may be set later by calling the `Cancel` operation, which registers the integer reason given as an argument.

Once the set of timers is established, it may be used through the `Timer_Expired` entry. This entry will suspend the calling task if the cancellation state is not set and if all of the calls to the `Timer_Expired` entry of the different timers attached to the set that are in the armed state would have blocked. If the cancellation state is set, the entry is allowed to complete and, in this case, it resets the cancellation state, and returns `True` in the `Cancelled` parameter and the reason registered by the last `Cancel` operation in the `Reason` parameter. Otherwise, if the cancellation state is not set and one or more of the armed timers attached to the sets has expired the entry call is allowed to complete, returning `False` in the `Cancelled` parameter and a reference to one of the timers that had expired; in this case, the `Reason` parameter is unspecified. If the set contains no timers the `Timer_Expired` entry can only complete when the cancellation state is set.

4. Example

The following code is an example of the structure of a monitoring task that lowers the priority of any of its monitored tasks that exceeds its execution time budget. First we define some constants and static objects, including the set of timers:

```
Max : constant Integer:=5;
Reason_Attach : constant Integer:=0;
Reason_Set_Budget : constant Integer:=1;

My_T : array (1..Max) of aliased Task_Id;
T_Set : Set_Of_Timers(Max);
```

The task specification contains two entries: `Attach_Task` is used to attach a new task to the set. `Set_Budget` is used to set the budget timer for a specific task in the set:

```
task Budget_Monitor is
   entry Attach_Task
     (T : Task_Id; Id : out Integer);
   entry Set_Budget
     (Id : Integer; Budget : Time_Span);
end Budget_Monitor;
```

The task's body invokes the `Timer_Expired` operation of the set of timers. Cancellation is used to report the budget monitor about a new task that requires attachment, or about a task that wants to set its execution time budget. The `Reason_Attach` and `Reason_Set_Budget` constants are respectively used for those purposes. After cancellation, the task accepts the appropriate call. If no cancellation had occurred, then one of the monitored tasks has overrun its budget, and in that case the monitor will lower its priority to a background level and report the error; of course, other appropriate actions could be taken at this point.

```
task body Budget_Monitor is
   The_Timer : array (1..Max) of Timer_Ref;
   Next_Budget : Execution_Time.CPU_Time;
   Num : Integer range 0..Max:=0;
   Canceled : Boolean;
   Expired : Timer_Ref;
   Reason : Integer;
begin
   loop
      T_Set.Timer_Expired
        (Canceled,Reason,Expired);
      if Canceled then
         case Reason is
            when Reason_Attach =>
               accept Attach_task
                 (T : Task_Id; Id : out Integer)
               do
                  Num:=Num+1;
                  My_T(Num):=T;
                  The_Timer(Num):= new
                    Execution_Time.Timer
                    (My_T(Num)'Access);
                  T_Set.Add(The_Timer(Num));
                  Id:=Num;
               end Attach_Task;
            when Reason_Set_Budget =>
               accept Set_Budget
                 (Id : Integer;
                  Budget : Time_Span)
               do
                  Next_Budget:=
                     Execution_Time.Clock
                       (My_T(Id))+Budget;
                  The_Timer(Id).Arm(Next_Budget);
               end Set_Budget;
            when others => null;
         end case;
      else
         -- lower the priority
         Set_Priority(System.Priority'First,
                      Expired.T.all);
         Report_Error;
      end if;
   end loop;
end Budget_Monitor;
```

The code of a task using the budget monitor is shown below:

```
task body Work is
   Budget : Time_Span:=<value>;
   Id : Integer;
   Next_Period : Real_Time.Time;
begin
   -- attach the task to the budget monitor
```

```
   T_Set.Cancel(Reason_Attach);
   Budget_Monitor.Attach_Task(Work'Identity,Id);
   loop
      -- set the budget
      T_Set.Cancel(Reason_Set_Budget);
      Budget_Monitor.Set_Budget(Id,Budget);
      -- do normal work
      Do_Useful_Work;
      Update_Next_Period;
      delay until Next_Period;
   end loop;
end Work;
```

We can see that after attaching itself to the budget monitor, the task enters a loop setting its budget, and later doing some useful work and waiting for its next period.

5. Implementation issues

The implementation of the set of timers and the multiple wait operation is relatively simple if POSIX execution time clocks and timers [4] are used as the underlying services for implementing the `Execution_Time` package. In POSIX, timers generate a signal when they expire; it is possible to simultaneously wait for a set of signals, through one of the `sigwait` functions. Cancellation is easy if an additional signal is used, or if one of the signals reserved for the timers is used with information attached to represent cancellation. A POSIX implementation of execution time clocks and timers is available in our free software kernel MaRTE OS [1][2].

In bare-machine implementations of `Execution_Time`, it is easy to connect the expiration of a timer with the change in the state of the barrier guarding the `Timer_Expired` entry.

6. Adding a Cancellation Entry to the Timers

We have seen that the `Cancel` operation is important to be able to interrupt the wait operation on the timers, for example to report a change in the state or the working conditions of the budget monitor. In fact, the same functionality would be interesting for a single timer, for example if the monitored task is aborted.

In this case we consider that the `Reason` parameter is not so useful as for the `Set_Of_Timers` protected objects, as the number of different reasons for interrupting a timer wait is probably not as high as in a multi-purpose task that is managing several budgets.

As a consequence, we propose adding the following entry and procedure to the `Timer` protected object (in addition to the existing ones):

```
entry Timer_Expired
  (Cancelled : out Boolean);
procedure Cancel;
```

The new `Timer_Expired` entry would be allowed to complete when the cancellation state of the object was set, in addition to the conditions defined by the parameterless `Timer_Expired` entry. It would return `True` in the `Cancelled` parameter if it had completed because of cancellation, and `False` otherwise. The call would reset the cancellation state. The `Cancel` operation would set the cancellation state.

7. Conclusion

The ability to manage several execution-time budgets from a single task is important to reduce the amount of tasks dedicated to monitoring activities, and to ease cooperative management. To include this functionality, in this paper we have proposed a simple extension to the `Execution_Time` package that is under consideration for the next revision of the Ada Language. The extension consists of a new protected type whose objects can store sets of execution-time timers. The set has an operation to simultaneously wait for one of many execution-time timers to expire. We have shown the applicability of this interface through a simple example.

References

[1] Aldea Rivas M. and González Harbour M. “MaRTE OS: Minimal Real-Time Operating System for Embedded Applications” Departamento de Electrónica y Computadores. Universidad de Cantabria. `http://marte.unican.es/`

[2] Aldea Rivas M. and González Harbour M. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, Lecture Notes in Computer Science, LNCS 2043, May, 2001, ISBN:3-540-42123-8, pp. 305,316.

[3] González-Harbour M., Aldea Rivas M., Gutiérrez García J.J., Palencia Gutiérrez J.C. “Implementing and using Execution-Time Clocks in Ada Hard Real-Time Applications”. International Conference on Reliable Software Technologies, Ada-Europa’98, Uppsala, Sweden, in Lecture Notes in Computer Science No. 1411, June, 1998, ISBN:3-540-64563-5, pp. 91,101.

[4] IEEE Std. 1003.1:2001, Information Technology —Portable Operating System Interface (POSIX).

[5] J. Miranda and M. González Harbour. “A Proposal to Integrate the POSIX Execution-Time Clocks into Ada 95”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2003, Tolouse, France, in Lecture Notes in Computer Science, LNCS 2655, June, 2003, ISBN 3-540-40376-0.

[6] AI95-00307/03 “Execution-time Clocks”, February 2003.