A Proposal to Integrate the POSIX Execution-Time Clocks into Ada 95*

Javier Miranda[1] and M. González Harbour[2]

[1] Applied Microelectronics Research Institute, Univ. Las Palmas de Gran Canaria
35017 Las Palmas de Gran Canaria, Spain
jmiranda@iuma.ulpgc.es
[2] Departamento de Electrónica y Computadores, Univ. Cantabria
39005 – Santander, Spain
mgh@unican.es

Abstract. In this paper we present a proposal to integrate the POSIX.1 execution-time clocks and execution-time timers into the Ada 95 language. This proposal defines a new package named *Ada.CPU_Time* and describes the modifications done to the GNAT front-end and run-time to support it. Additionally this proposal discusses some usage schemes of this new interface.

Keywords: Scheduling, Hard Real-Time, Ada 95, Execution-Time, GNAT

1 Introduction

Traditional real-time systems were built (and still are) using executive schedulers [4]. In these systems, if a particular task or routine exceeded its budgeted execution time, the system could detect the situation. Basically, whenever the minor cycle interrupt came in, it could check whether the current action had completed or not. If not, that meant an overrun. Unfortunately, in concurrent real-time systems built with multitasking preemptive schedulers, there is no equivalent method to detect and handle execution-time overruns. This is the case for systems built using the Ada tasking model and the associated Real-Time Annex [9].

In addition to detecting budget overruns, 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 is essential to be able to implement such flexible scheduling algorithms.

* This work has been partially funded by the Spanish Research Council (CICYT), contract numbers TIC2001–1586–C03–03 and TIC99-1043-C03-03, and by the Commission of the European Communities under project IST-2001-34820 (FIRST).

J.-P. Rosen and A. Strohmeier (Eds.): Ada-Europe 2003, LNCS 2655, pp. 344–358, 2003.

In recognition of all these application requirements, the Real-Time extensions to POSIX have recently incorporated support for them. Real-Time POSIX supports execution-time clocks and timers that allow an application to monitor the consumption of execution time by its tasks, and to set limits for this consumption. The next revision of the Ada language should support this functionality in order for the language to continue to be the best for programming real-time applications.

There has been a previous proposal to include execution-time clocks and timers into the Ada language [6,3]. That proposal defines a new package that includes all the operations required to access the execution-time monitoring functionality. Some of those operations belong to a protected object type that represents execution-time timers. That approach does not require modification of the compiler, nor of the run-time system (provided that the POSIX execution-time functionality is available), but its execution-time type is not integrated in the Ada language as a Time type.

In this paper we present a proposal to integrate the POSIX.1 execution-time clocks and timers into the Ada 95 language. In this proposal we try to minimize the implementation impact on the compiler and run-time system, and therefore we will not include any new language construct. It is composed of a new package named *Ada.CPU_Time* and the proposed modifications to the GNAT [5] front-end and run-time to efficiently support the execution-time clocks and timers. The proposal is being submitted to the Ada Rapporteur Group (ARG) for possible inclusion in a future revision of the Ada standard.

This paper is organized as follows. In Sect. 2 we present the interface of our proposed *Ada.CPU_Time* Ada package. In Sect. 3 we present five basic usage schemes of the execution-time clocks and timers. In Sect. 4 we briefly present the modifications done to the GNAT compiler to allow the use of execution-time timers in the Ada 95 timed sentences. We close with some conclusions and references.

2 Ada.CPU_Time

Our proposed interface to handle CPU_Time is based on the standard Ada.Real_Time interface. The major differences between our Ada.CPU_Time interface and the standard Ada.Real_Time interface are:

- The new data type *Clock_Id* is used to represent execution-time clocks. A value of this type represents the execution-time clock of a given task. According to the POSIX definition of execution time [7], it is implementation defined to whom will be charged the effects of interrupt handlers and run-time services on behalf of the system.
- The *Time* type represents absolute values of execution time as measured by a given execution-time clock. Values of this type have an internal Clock_Id value that ties the value of execution time to its associated clock. If a variable of type Time is not initialized, the value of its internal Clock_Id is undefined.

The type Time is a time type as defined by ARM95, Sect. 9.6, and thus values of this type may be used in a delay_until_statement.
- The *Time_Span* type represents length of execution-time duration, and its values are not dependent upon any particular execution-time clock (or task).
- The new function *CPU_Clock* is used to get the execution-time clock identifier associated with each Ada task.
- The function *Clock* has a new parameter used to specify the execution-time clock to be read. In order to keep compatibility with the Ada *Calendar* and *Real_Time* packages, if no parameter is passed then the execution-time clock of the calling task is returned.
- The new function *Clock_Id_Of* returns the identifier of the execution-time clock associated with the Time parameter T.
- The function *Time_Of* has a new parameter used to associate the time to an execution-time clock.
- The exception *Time_Error* is raised by the function Clock if the Clock_Id parameter is not valid. This exception is also raised by operators "+" and "-", and the function Clock_Id_Of, if an execution-time parameter is not valid (for example, if it is not initialized). In addition, it is also raised by a delay_until_statement if a Time value corresponding to the task executing the statement is used, because otherwise this situation would cause a deadlock.
- The exception *Incompatible_Times* is raised by operator "-" if the execution-time parameters correspond to different execution-time clocks.

```
with Ada.Task_Identification;
package Ada.CPU_Time is
   type Clock_Id is private;

   type Time is private;
   Time_First : constant Time;
   Time_Last  : constant Time;
   Time_Unit  : constant := implementation-defined-real-number;

   type Time_Span is private;
   Time_Span_First :  constant Time_Span;
   Time_Span_Last  :  constant Time_Span;
   Time_Span_Zero  :  constant Time_Span;
   Time_Span_Unit  :  constant Time_Span;

   Tick : constant Time_Span;

   function CPU_Clock
     (Task_Id : Ada.Task_Identification.Task_Id
        := Ada.Task_Identification.Current_Task) return Clock_Id;
   function Clock (C : Clock_Id) return Time;
   function Clock_Id_Of (T : Time) return Clock_Id;

   function "+"  (Left : Time;      Right : Time_Span) return Time;
   function "+"  (Left : Time_Span; Right : Time)      return Time;
   function "-"  (Left : Time;      Right : Time_Span) return Time;
   function "-"  (Left : Time;      Right : Time)      re-
turn Time_Span;
   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
```

```
     function ">"  (Left, Right : Time) return Boolean;
     function ">=" (Left, Right : Time) return Boolean;
     function "+"  (Left, Right : Time_Span) return Time_Span;
     function "-"  (Left, Right : Time_Span) return Time_Span;
     function "-"  (Right : Time_Span) return Time_Span;
     function "*"  (Left  : Time_Span; Right : Integer) re-
turn Time_Span;
     function "*"  (Left  : Integer;   Right : Time_Span) re-
turn Time_Span;
     function "/"  (Left, Right : Time_Span) return Integer;
     function "/"  (Left : Time_Span; Right : Integer) return Time_Span;
     function "abs" (Right : Time_Span) return Time_Span;
     function "<"  (Left, Right : Time_Span) return Boolean;
     function "<=" (Left, Right : Time_Span) return Boolean;
     function ">"  (Left, Right : Time_Span) return Boolean;
     function ">=" (Left, Right : Time_Span) return Boolean;

     function To_Duration  (TS : Time_Span) return Duration;
     function To_Time_Span (D : Duration)   return Time_Span;
     function Nanoseconds  (NS : Integer) return Time_Span;
     function Microseconds (US : Integer) return Time_Span;
     function Milliseconds (MS : Integer) return Time_Span;

     type Seconds_Count is new Integer range -Integer'Last ..
              Integer'Last;
     procedure Split (T : Time; SC : out Seconds_Count; TS :
              out Time_Span);
     function Time_Of (SC: Seconds_Count; TS : Time_Span; C : Clock_Id)
        return Time;

     Time_Error         : exception;
     Incompatible_Times : exception;
  private
     . . .
  end Ada.CPU_Time;
```

3 Usage Schemes for Ada.CPU_Time Timers

Using package *Ada.CPU_Time*, described above, we can design some basic usage schemes that depend on the particular needs of the application task whose execution time is being monitored. We have identified five major schemes:

- **Handled** [6]. This is the case in which an execution-time overrun is detected, but the task is allowed to complete its execution. This is applicable to systems under testing, or for tasks that have a high degree of critically (an thus cannot be stopped) or for which an occasional execution-time overrun can be tolerated, but needs to be reported.

  In this scheme, the application task uses a single variable T to remember the value of the execution-time clock and to evaluate the execution time of the work. If the execution time is higher than MAX_TIME then it handles the execution-time error.

```
task body Periodic_Handled is
   C : Clock_ID := CPU_Clock; --  My execution-time clock identifier.
   T : Time;
begin
   loop
      T := Clock (C);
      do task useful work;
      if Clock (C) - T > MAX_TIME then
         Handle the error;
      end if;
      delay until Next_Start; --  Global clock delay.
   end loop;
end Periodic_Handled;
```

– **Priority Change** [6]. In this scheme when the overrun is detected the priority of the task is lowered or increased (depending on the application requirements). A simple implementation of this scheme uses two nested tasks: the *Worker* task and the *Supervisor* task. The Supervisor task sleeps until the execution time of the Worker task reaches the instant of the priority change. When this happens the supervisor lowers (or increases) the priority of the worker task. If the Worker task completes the work before the instant of the priority change then it aborts the Supervisor task.

```
task body Worker is
   task Supervisor;
   task body Supervisor is
      C : Clock_ID := CPU_Clock (Worker'Identity);
      T : Time := Clock (C) + To_Time_Span (TIME_OF_PRIORITY_CHANGE);
   begin
      delay until T;
      Lower (or increase) the priority of the worker task;
   end Supervisor;
begin
   do useful work;
   abort Supervisor;
end Worker;
```

An alternative implementation of this scheme does not require *abort*. It can be done by means of the Ada *select* statement and one *entry* (i.e, *Work_Done*). When the worker completes its work calls Work_Done. The supervisor can then be implemented by means of a timed selective accept. If the entry call is received before the instant of priority change is reached then the work has been successfully completed in time; otherwise the time-budget has expired and the Supervisor lowers (or increases) the priority of the worker task. In addition the Supervisor task must accept the call to Work_Done that will be issued by the Worker at the end of its work.

```
task Worker;

task Supervisor is
   entry Work_Done;
end Supervisor;

task body Worker is
```

```
begin
   do useful work;
   Supervisor.Work_Done;
end Worker;

task body Supervisor is
  C : Clock_ID := CPU_Clock (Worker'Identity);
  T : Time := Clock (C) + To_Time_Span (TIME_OF_PRIORITY_CHANGE);
begin
  select
     accept Work_Done;
  or
     delay until T;
     Lower (or increase) the priority of the worker task;
     accept Work_Done;
  end select;
end Supervisor;
```

– **Stopped** [6]. This is the case in which if an execution-time overrun is detected, the associated task execution is stopped to allow lower priority tasks to execute within their deadlines. The task itself waits until its next activation and then proceeds normally.
In the implementation of this scheme an asynchronous select statement is used to abort the task's work if an execution-time overrun is detected.

```
task body Periodic_Stopped is
   C : Clock_ID := CPU_Clock;  --  My execution-time clock identifier
   T : Time;
   Next_Start : Duration;
begin
   loop
      T := Clock (C) + To_Time_Span (WORST_CASE_EXEC_TIME);
      select
         delay until T;         --  Execution-time clock
         Handle the error;
      then abort
         do useful work;
      end select;
      Next_Start := ...;
      delay until Next_Start;  --  Global clock
   end loop;
end Periodic_Stopped;
```

– **Imprecise** [6]. This scheme corresponds to the case in which the task is designed using the imprecise computation model [10], in which the task has a mandatory part (generally short and for which it is easier to estimate a worst-case execution time), and an optional part that refines the calculations made by the task. Since the worst-case execution time of this optional part is usually more difficult to estimate, this part will be aborted if an execution-time overrun is detected. This technique is also valid for cases in which the optional part continuously refines the quality of the results; we can let the optional part run until it exhausts its execution-time budget, and then use the last valid result obtained.

The implementation of this scheme consists of using the "handled" approach for the mandatory part of the task, and the "stopped" approach for the optional part. After the optional part, whether it is aborted or not, another mandatory part may exist to cause outputs of the task to be generated. Therefore this scheme can be implemented as follows:

```
task body Periodic_Imprecise is
   C : Clock_ID := CPU_Clock;  --  My execution-time clock identifier
   T : Time;
   Next_Start : Duration;
begin
   loop
      T := Clock (C) + To_Time_Span (WORST_CASE_EXEC_TIME);
      do mandatory part I;
      select
         delay until T;         --  Execution-time clock
      then abort
         do optional part;
      end select;
      do mandatory part II;
      Next_Start := ...;
      delay until Next_Start;  --  Global clock
   end loop;
end Periodic_Imprecise;
```

4 Detailed Description of the Integration of the Execution-Time Timers into the GNAT Compiler

In this section we describe the modifications done to the GNAT compiler to support the POSIX execution-time timers in Ada. For each Ada timing statement (delay until, timed entry call, and timed selective accept) we present the modifications done to the GNAT front-end and run-time.

4.1 Delay Until Statement

Front-End

- **Semantics.** The subprogram *Analyze_Delay_Until* has been modified to allow the use of the *Ada.CPU_Time.Time* type in the Ada 95 *delay until* statement.
- **Expander.** When an *Ada.CPU_Time.Time* type variable is used to specify the timeout, the expander has been modified to transform the *delay until* statement as follows:

```
Original Ada Code              Expanded Code
-----------------              -------------
delay until T;                  Ada.CPU_Time.Delays.Delay_Until (T);
```

Run-Time

- Package **System.Tasking.** A POSIX execution-time timer and two flags have been added to the Ada Task Control Block (ATCB). The flags are used to remember if the timer has been created (and therefore the execution-time timer field is valid), and if the ATCB timer is currently in use.
- Package **Ada.CPU_Time.Delays.** The subprogram *Delay_Until* has been programmed to do the following actions:
  1. Defer the abortion of the calling task.
  2. Lock the ATCB of the calling task.
  3. If the ATCB has not been created then create, program, and arm the ATCB execution-time timer; if the ATCB had been created but it is not in use then program and arm the ATCB execution-time timer; otherwise create, program and arm a new execution-time timer. In all these cases the address of a *Delay Block* register composed of the following fields is associated with the execution-time timer:
     - The ATCB address of the calling task.
     - A boolean field (*Timed_Out*) initialized to false. This field will be used by the timed statements to differentiate the case of the timeout expiration from the case in which the blocked task is awakened by some other task (i.e. the acceptor of a timed entry call, or the caller of a selective wait).
  4. Pass the calling task to the *Delay_Sleep* state.
  5. Stop the calling task until the timeout expires. This is done by blocking the calling task using the caller *mutex* and *condition variable* declared by GNAT in the ATCB for this purpose.
  6. Pass the calling task to the *Runnable* state.
  7. If the ATCB timer was re-used then mark it as "not in use". Otherwise, remove the execution-time timer.
  8. Unlock the ATCB of the calling task.
  9. Verify if a request to abort the calling task has been received during this delay. If true then abort the task; otherwise undefer its abortion.

  A task is used to handle the signal associated with all the execution-time timers. This task does following actions:
  1. Block all the signals.
  2. Activate the signal associated with all the execution-time timers.
  3. Await for the execution-time timers signal.
  4. Get the address of the *Delay Block* register associated with the execution-time timer.
  5. Set to *True* the field *Timed_Out* of this *Delay Block* register.
  6. Awaken the task that programmed this execution-time timer.
  7. Go to step 3.

Behavior. The calling task programs an execution-time timer and becomes blocked until this timer expires.

4.2 Timed Entry Call

Front-End

- **Semantics.** No modification was required.
- **Expander.** When an *Ada.CPU_Time.Time* variable is used to specify the timeout the expander has been modified to transform the Ada timed entry call statement in the following way:

```
Original Ada Code              Expanded Code
-----------------              -------------
select                           declare
  T.E                               P : params := (param, param, param);
  <<S1>>;                           B : Boolean;
or                                begin
  delay until <<CPU_TIMEOUT>>;      CPU_Timed_Entry_Call
  <<S2>>                              (Acceptor => <Acceptor-Task_ID>,
end select;                            Entry_Id => <Entry_Index>,
                                        Uninterpreted_Data => P'Address,
                                        Timeout  => <<CPU_TIMEOUT>>,
                                        Mode     => Absolute_CPU_Mode,
                                        Successful => B);
                                    if B then
                                      <<S1>>;
                                    else
                                      <<S2>>;
                                    end if;
                                  end;
```

Run-Time

- Package **System.Tasking.Rendezvous** The new *CPU_Timed_Entry_Call* subprogram is based on the GNAT *Timed_Entry_Call subprogram*. The main differences with the original GNAT version are:
  - The data type of the *Timeout* parameter is *Ada.CPU_Time.Time* (instead of *Duration*).
  - Its body calls *CPU_Wait_For_Completion_With_Timeout* instead of the GNAT *Wait_For_Completion_With_Timeout* version.
- Package **System.Tasking.Entry_Calls** The new subprogram *CPU_Wait_For_Completion_With_Timeout* is based on the GNAT subprogram *Wait_For_Completion_With_Timeout*. The only difference is that it calls *CPU_Timed_Sleep* (instead of the GNAT subprogram *Timed_Sleep*).
- Package **System.Task_Primitives.Operations** The new subprogram *Ada.CPU_Timed_Sleep* calls *Ada.CPU_Time.Delays.Delay_Until*.

Behavior. If the rendezvous can be immediately accepted the subprogram *CPU_Timed_Entry_Call* completes the rendezvous and returns *True* in the out mode parameter *Successful*. Otherwise it programs an execution-time timer by calling the subprogram *Ada.CPU_Time.Delays.Delay_Until*.

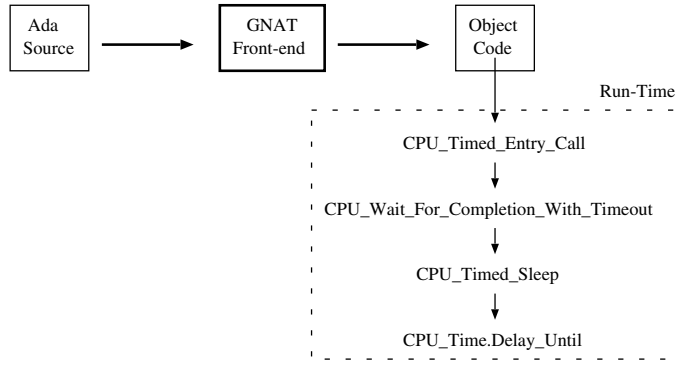

Fig. 1. Run-time calls to implement the CPU timed entry call

- If the call is accepted before the timeout expires then the receiver task unblocks the caller by calling *Wakeup*, the same subprogram called by the CPU_Time signal handler. The unblocked calling task detects this state by evaluating the *Signaled* field associated with its execution-time timer (still false because the execution-time timeout has not expired). Therefore the calling task removes its execution-time timer and returns *True* in its out mode parameter *Successful*.
- Otherwise (the timeout expires) the CPU_Time signal handler sets to *True* the Signaled field associated with the execution-time timeout, and the unblocked calling task removes its execution-time timer and returns *False* in its out mode parameter *Successful*.

4.3 Timed ATC

Front-End

- **Semantics.** No modification required.
- **Expander.** When the Ada.CPU_Time time variable is used to specify the timeout the GNAT expander has been modified to generate the following block of code:

```
Original Ada Code
-----------------
select
  delay until <T>;
  <<S1>>;
or
  <<abortable_part>>
end select;
```

```
Expanded Code
-------------
declare
   D : aliased Delay_Block;
begin
   Abort_Defer;
   CPU_Arm_Timer (<T>, D'access);
   begin
     begin
       Abort_Undefer;
       <<abortable_part>>
     at end
       Async_Cancel_Timer (D'access);
```

```
      end;
    exception
      when _abort_signal =>
         Abort_Undefer;
    end;
    if Timed_Out (D) then
      <<S1>>;
    end if;
  end;
```

Run-Time

- Package **System.Tasking.Async_Delays.** The new CPU_Arm_Timer subprogram does the following actions:
  1. Increments the ATC nesting level of the calling task.
  2. Initializes all the fields of the Delay_Block.
  3. Calls the subprogram *Ada.CPU_Time.Delays.Arm_Timer*.
- Package **Ada.CPU_Time.Delays.** The new CPU_Arm_Timer subprogram arms the execution-time timer and returns.

Behavior. First of all let's briefly explain the semantics of the GNAT "at end" statement. It is a handler which provides a common way out of a block of statements even when an exception is propagated.

In the above code, after the execution-time timer is armed the abortable part is executed. If the abortable part completes its execution before the execution-time timer expires then the *Async_Cancel_Timer* is called to disarm the timer. Otherwise the execution-time timer expires and the signal catcher calls the run-time subprogram *Locked_Abort_To_Level* which defers the abortion of the blocked task, and cancels all the nested ATC (if any) done in the abortable part by raising the internal exception *_abort_signal*. After the abortion is undefered (in the exception handler) if the timeout had expired then the block of statements $<< S1 >>$ is executed.

4.4 Timed Selective Accept

Front-End

- **Semantics.** It has been modified to disallow the simultaneous use of *Ada.CPU_Time* and *Ada.Real_Time* time variables to specify multiple timeouts in the Ada 95 timed selective accept[1].
- **Expander.** When the Ada.CPU_Time time variable is used to specify the timeouts of the following Ada 95 timed-selective statement the GNAT expander has been modified to generate the following block of code:

[1] "If a selective_accept contains more than one delay_alternative, then all shall be delay_relative_statements, or all shall be delay_until_statements for the same time type." ARM95 [9], Sect. 9.7.1(13).

```
Original Ada Code
--------------
select
  accept E1 ...;
or
  accept EN ...;
or
  delay until <CPU_TIMEOUT_1>;
or
  delay until <CPU_TIMEOUT_N>;
end select;


Expanded Code
-------------
declare
   S : Entry_Barriers := (others => True);
   P : params := (param, param, param);
   D : time_array (1 .. N) :=  (<CPU_TIMEOUT_1>, <CPU_TIMEOUT_N>);
   E_Index : integer := 0;
   D_Index : integer := 0;
begin
    CPU_Timed_Selective_Wait
        (Open_Accepts       => S'address,
         Select_Mode        => delay_mode,
         Uninterpreted_Data => P'Address,
         Timeout            => D,
         Mode               => Absolute_CPU_Mode,
         Index              => E_Index,
         CPU_Time_Index     => D_Index);
    if E_Index = 0 then
         -- Some timeout has expired.
         case D_Index is
              when 1 => <<CPU_S1>>
              ...
              when N => <<CPU_SN>>
         end case;
    else
         -- Some entry call has
         -- been accepted.
         case E_Index is
              when 1 => <<S1>>
              ...
              when N => <<SN>>
         end case;
    end if;
end;
```

Run-Time

– Package **System.Tasking.Rendezvous.** The new *CPU_Timed_Selective_Wait* subprogram is based on the GNAT *Timed_Selective_Wait* subprogram. The main differences with the original GNAT version are:

  - The data type of the *Timeout* parameter is an array where all the timeouts specified in the Ada *select* statement are passed by the front-end.

- Its body calls a variant of the *CPU_Timed_Sleep* which receives the timeouts array and returns the index of the expired CPU timeout. This index is returned in the *CPU_Time_Index* parameter. The possible values of the out mode parameters *Index* and *CPU_Time_Index* are:

```
                                        Index          CPU_Time_Index
                                        -----          --------------
Some entry call was accepted:       <entry index>           0
Some deadline expired:                    0             <timeout index>
```

– Package **System.Task_Primitives.Operations.** The new variant of the subprogram *Ada.CPU_Timed_Sleep* calls *Ada.CPU_Time.Delays.Multiple_Delay_Until*.
– Package **Ada.CPU_Time.Delays.Multiple_Delay_Until.** This subprogram does the same actions as *Delay_Until* (described in Sect. 4.1). However, instead of using a single execution-time timer, it programs as many execution-time timers as the number of *delay until* alternatives specified by the programmer in the Ada 95 timed selective accept.
In order to identify the execution-time timer which expired, an array of *Delay Block* registers containing the address of the caller's ATCB and the *Timed_Out* field is used. When one execution-time timeout expires the unblocked task traverses this array to look for the execution-time timeout which has its *Timed_Out* field set to *True*. If no execution-time timer has its *Timed_Out* field set to true it means that some entry call was accepted, and therefore the task was unblocked by the caller.

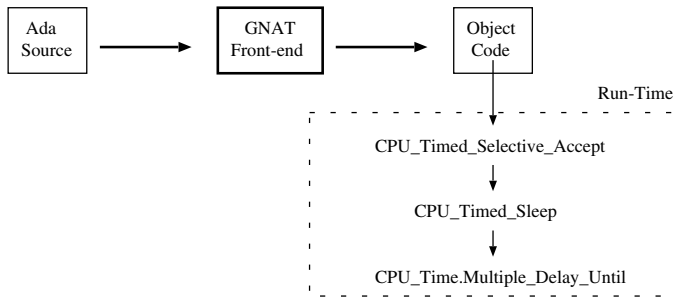

Fig. 2. Run-time calls to implement the CPU timed selective accept

5 Discussion

A prototype implementation has been developed for the execution-time budgeting proposal presented in this paper, using the MaRTE operating system [1,2] that provides a POSIX.13 [8] minimal real-time system interface and includes

execution-time clocks and timers. As we have shown, the implementation requires small modifications to the compiler and to the run-time. The modifications were implemented in a short period of time and should not represent a large effort to current compiler implementors. We have compared this implementation with the one presented in [3] defining support for the execution-time clocks and timers as a library package. This latter implementation is very simple, because it does not require modifications to the compiler nor to the runtime system. The overheads are quite similar in both implementations, and are small.

For both approaches, implementations on bare machines or systems without the POSIX execution-time clocks and timers would be a bit more complex because the underlaying execution-time monitoring functionality would have to be implemented in the scheduler. Reference [6] describes one such implementation and it can be seen that it is relatively simple, and that it does not introduce any significant overhead into the scheduler.

Both approaches were discussed at the International Real-Time Ada Workshop in 2002. The conclusion from the Workshop was to recommend the approach presented in this paper, with execution-time functionality integrated in the Ada language, because it provides a simpler model to programmers. However, the group felt that there is a strong need to have the execution-time functionality in Ada, so if the second proposal, with a library implementation, has more probability of success, the group would also recommend its adoption.

Both proposals have been submitted to the Ada Rapporteur Group (ARG) for the inclusion of the execution-time clocks in the next revision of the Ada language. At the time of writing this paper the ARG has expressed preference for the package solution because it has less implementation impact, but a final decision has not been made, and discussions will continue.

6 Conclusions

We have presented a proposal to integrate the POSIX.1 execution-time clocks and timers into the Ada 95 language. This proposal defines a new package named *Ada.CPU_Time* and describes the modifications done to the GNAT front-end and run-time to allow the use of the execution-time timers in the timed Ada statements. We have also discussed some usage schemes of the *Ada.CPU_Time* interface.

As a proof of concepts we have modified the GNAT sources to support the execution-time timers with all the Ada 95 timed statements: delay until, timed entry call (to tasks and to protected objects), timed asynchronous transfer of control (ATC). and timed selective accept. We have a modified version of the GNAT compiler which implements all the proposals presented in this paper, and which can be used on top of the MaRTE Operating System [1,2]. Among the different options for implementing execution-time clocks in the Ada language, this proposal represents an easy to use alternative with limited implementation impact. It is now the task of the ARG and the Ada community to decide which of the alternatives is best for inclusion in the next revision of the Ada language.

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. Aldea Rivas M. and González Harbour M. *Extending Ada's Real-Time Systems Annex with the POSIX Scheduling Services*. IRTAW-2000, Las Navas, Avila, Spain.
4. Burns A. and Wellings A. *Real-Time Systems and Programming Languages*. 3rd edition. Addison-Wesley, 2001.
5. Comar, C., Gasperoni, F., and Schonberg, E. *The GNAT Project: A GNU-Ada9X Compiler*. Technical report. New York University. 1994.
6. 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.
7. IEEE Std. 1003.1:2001, Information Technology – Portable Operating System Interface (POSIX).
8. IEEE Std. P1003.13-1998, Information Technology – Standarized Application Environment Profile – POSIX Realtime Application Support (AEP). The Institute of Electrical and Electronics Engineers.
9. Intermetrics, Inc. *Ada 95 Language Reference Manual*. Intermetrics, Inc., Cambridge, Mass., January, 1995.
10. J. Liu, K.J. Lin, W.K. Shih, A. Chang-Shi Yu, J.Y. Chung, and W. Zhao. *Algorithms for Scheduling Imprecise Computations*. IEEE Computer, pp. 56–68, May 1991.