Ada Issue 307 Execution-Time Clocks

!standard D.14 (00) 05-03-16 AI95-00307/11
!standard D.14.1 (00)
!class amendment 02-08-28
!status Amendment 200Y 04-06-24
!status ARG Approved 8-0-0 04-06-13
!status work item 03-09-21
!status ARG Approved 12-0-1 03-06-20
!status work item 02-08-28
!status received 02-08-28
!priority High
!difficulty Hard
!subject Execution-Time Clocks

!summary

Modern real-time scheduling policies require that applications have the ability to measure the execution time of tasks, and to detect execution time overruns. This functionality is provided by a new package that defines execution time clocks and a child of this package which provides timers.

!problem

Performance analysis techniques are usually based on the assumption that the application developer can accurately measure/estimate the execution time of each task. Measurement is always very difficult, because, with effects like cache misses, pipelined and superscalar processor architectures, etc., the execution time is highly unpredictable. There are models that allow the calculation of worst-case execution time (WCET) for some architectures, but they are generally very complex and not widely available for all architectures. A language defined means of measuring execution time is desirable.

In hard real-time systems it is essential to monitor the execution times of all tasks and detect situations in which the estimated WCET is exceeded. This detection is usually available in systems scheduled with cyclic executives, because the periodic nature of a cycle allows checking that all initiated work has 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 would ease implementation of such flexible scheduling algorithms.

The Real-Time extensions to POSIX have recently incorporated support for execution time monitoring. 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.

It could be argued that given that the POSIX standard already defines execution time clocks and timers it would not be necessary to have the same functionality in the Ada language standard, because the POSIX services can be accessed through the appropriate bindings. This is true for Ada platforms built on top of POSIX OS implementations, but not for bare-machine implementations, like the ones used in embedded systems, avionics, etc. For portability purposes it is necessary to have this functionality supported in a homogeneous way for all the implementations that choose to support it.

!proposal

(See wording.)

!wording

Add new section D.14.

D.14 Execution Time

This clause describes a language-defined package to measure execution time.

Static Semantics

The following language-defined library package exists:

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.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 : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

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

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

The *execution time* or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. It is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers and run-time services on behalf of the system.

The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers.

The CPU_Time value I represents the half-open execution-time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation-defined real number. For each task, the execution time value is set to zero at some unspecified point between the creation of the task and the start of the activation of the task.

CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

Dynamic Semantics

CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A *CPU clock tick* is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals.

The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types.

The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 + TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit.

Erroneous Execution

For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous.

Implementation Requirements

The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task creation to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

Documentation Requirements

The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

The implementation shall document the properties of the underlying mechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

Metrics

The implementation shall document the following metrics:

- An upper bound on the execution-time duration of a clock tick. This is a value D such that if t1 and t2 are any execution times of a given task such that t1<t2 and Clock[t1]=Clock[t2] then t2-t1 <= D.

- An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution

-time clock (as observed by calling the Clock function with the same Task_Id).

- An upper bound on the execution time of a call to the Clock function, in processor clock cycles.

- Upper bounds on the execution times of the operators of the type CPU_Time, in processor clock cycles.

Implementation Permissions

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the CPU_Time type.

Implementation Advice

When appropriate, implementations should provide configuration mechanisms to change the value of CPU_Tick.

Add new section D.14.1:

D.14.1 Execution Time Timers

This clause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time.

Static Semantics

The following language-defined library package exists:

```
with System;
package Ada.Execution_Time.Timers is

   type Timer (T : access Ada.Task_Identification.Task_Id) is
      limited private;

   type Timer_Handler is access protected procedure (TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority :=
    implementation-defined;

   procedure Set_Handler (TM      : in out Timer;
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   function Current_Handler (TM: Timer) return Timer_Handler;
   procedure Cancel_Handler (TM        : in out Timer;
                             Cancelled : out Boolean);

   function Time_Remaining (TM : Timer) return Time_Span;

   Timer_Resource_Error : exception;
```

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

The type Timer represents an execution-time event for a single task and is capable of detecting execution time overruns. The access discriminant T identifies the task concerned. The type Timer needs finalization (see 7.6).

An object of type Timer is said to be *set* if it is associated with a non-null value of type Timer_Handler and *cleared* otherwise. All Timer objects are initially cleared.

The type Timer_Handler identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a *handler*.

Dynamic Semantics

When Timer objects is created, or upon the first call of a Set_Handler procedure with the timer as parameter, the resources required to operate an execution-time timer based on the associated execution-time clock are allocated and initialized. If this operation would exceed the available resources, Timer_Resource_Error is raised.

The procedures Set_Handler associate the handler Handler with the timer TM; if Handler is null, the timer is cleared, otherwise it is set. The first procedure Set_Handler loads the timer TM with an interval specified by the Time_Span parameter. In this mode, when the execution time of the task identified by TM.T has increased by In_Time, the timer TM is said to have expired. The second procedure Set_Handler loads the timer TM with the absolute value specified by At_Time. In this mode, when the the execution time of the task identified by TM.T reaches At_Time, the timer TM is said to have *expired*; if the value of At_Time has already been reached when Set_Handler is called, the timer TM is said to be expired.

A call of a procedure Set_Handler for a timer that is already set replaces the handler and the (absolute or relative) execution time; if Handler is not null, the timer remains set.

When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

The function Current_Handler returns the handler associated with the timer TM if that timer is set; otherwise it returns null.

The procedure Cancel_Handler clears the timer if it is set. Cancelled is assigned True if the timer was set prior to it being cleared; otherwise it is assigned False.

The function Time_Remaining returns the execution time interval that remains until the timer TM would expire, if that timer is set; otherwise it returns Time_Span_Zero.

The constant Min_Handler_Ceiling is the priority value that ensures that no ceiling violation would occur, were a handler to be executed.

For all the subprograms defined in this package, Tasking_Error is raised if the task identified by TM.T has terminated, and Program_Error is raised if the value of TM.T is Task_Identification.Null_Task_Id.

An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

Erroneous Execution

For a call of any of the subprograms defined in this package, if the task identified by TM.T no longer exists, the execution of the program is erroneous.

Implementation Requirements

For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

AARM note:
This prevents various race conditions. In particular it ensures that if an event occurs when Set_Handler is changing the handler then either the new or old handler is executed in response to the appropriate event. It is never possible for a new handler to be executed in response to an old event.
end AARM note

When an object of type Timer is finalized, the system resources used by the timer shall be deallocated.

Implementation Permissions

Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded then Timer_Resource_Error is raised.

Notes

A Timer_Handler can be associated with several Timer objects.

!discussion

Modern real-time scheduling policies require that applications have the ability to measure execution time of tasks, and to detect execution time overruns. Two mechanisms were proposed to achieve these requirements. One was based on a library-level package, and the other one was integrated into the Ada 95 language by creating a new Time type that could be used in the languages time constructs such as the delay until statement or various forms of the select statement. Prototype implementations exist for both.

After discussion by the ARG it was decided that the package mechanism was preferable.

A revision of the AI was produced by IRTAW 12 in response to the requirement to be able to wait for one or more Timers to expire. This has required a restructuring of the earlier solution. The consequence is that the Timer is no longer a protected object. Timer expirations are notified by the system by calling a protected procedure that is passed to it by a parameter in the Timer operations.

A prototype implementations was developed for the initial proposal, using the MaRTE operating system that provides a POSIX.13 interface and includes execution-time clocks and timers.

The implementation of the proposal is very simple, because it does not require modifications to the compiler nor to the runtime system.

Implementations on bare machines or systems without POSIX execution-time clocks and timers would be a bit more complex because the underlying execution-time monitoring functionality would have to be implemented in the scheduler. Document [2] 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.

The overhead of the implementation is small.

The original proposal included Time_Of taking a Task_Id parameter. This was included so that an implementation could choose to have the Task_Id as part of an Execution_Time value, in case we wished to

disallow operations between CPU_Time values of different tasks. Since we allow operations between CPU_Time values from different tasks, the Task_Id is not needed.

The initial proposal left the initial value of the execution time clock of each task unspecified. However, it is valuable to be able to determine a task's absolute execution time, and thus the value should be forced to start at zero. In some implementations this requirement may require saving an initial value of the CPU-Time clock, but this isn't an expensive requirement.

Because the execution time value may be set to zero during the task's activation, the requirement is that the value is set to zero at some unspecified point between the task creation and the end of the task's activation.

The initial proposal had two protected operations inside the Timer: Initialize and Finalize. Initialize took a parameter that identified the execution time clock to be used. These operations appeared unnecessary

to reviewers; the parameter could be moved to an access discriminant of the protected type or a parameter of the Set_Handler operation. The reviewers preferred the later solution.

Without these operations, a library implementation of the package on top of the POSIX execution-time services will have to include a component derived from Finalization. Controlled in the protected type to ensure that appropriate initialization and finalization are made. This should not cause a hardship; restricted runtime implementations may have to resort to some "magic", but most implementations will not need extra support. In either case, the overhead is small and bounded.

An earlier version of the timer had procedures Arm and Disarm and three states, armed, armed but expired, and disarmed. It was changed to bring it into line with the style of AI-297 on real-time timers.

!example

Example 1: Stopped task

The following is an example of a periodic task that limits its own execution time to some predefined amount called WCET (worst-case execution time). If an execution time overrun is detected, the task aborts the remainder of its execution, until the next period.

```
with Ada.Real_Time, Ada.Execution_Time.Timers, Ada.Task_Identification;
use Ada;
use type Ada.Real_Time.Time;

...

   task Periodic_Stopped;

   protected Control is
     entry Wait_Budget_Expiry;
     procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer);
     pragma Priority(Ada.Execution_Time.Timers.Min_Handler_Ceiling);
   private
     Expired : Boolean := False;
   end Control;

   protected body Control is
     entry Wait_Budget_Expiry when Expired is
     begin
       Expired := False;
```

```
    end Wait_Budget_Expiry;

    procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer) is
    begin
      Expired := True;
    end Budget_Expired;
  end Control;

  task body Periodic_Stopped is
    My_Id : aliased Task_Identification.Task_Id:=
      Periodic_Stopped'Identity;
    The_Timer : Ada.Execution_Time.Timers.Timer(My_Id'Access);

    Next_Start : Real_Time.Time:=Real_Time.Clock;
    WCET : constant Duration:=1.0E-3;
    Period : constant Duration:=1.0E-2;
  begin
    loop
      Ada.Execution_Time.Timers.Set_Handler
        (The_Timer, Real_Time.To_Time_Span(WCET),
         Control.Budget_Expired'Access);
      select
        Control.Wait_Budget_Expiry; -- Execution-time overrun detection
        -- Handle_the_error;
      then abort
        null; -- Do_useful_work;
      end select;
      Ada.Execution_Time.Timers.Cancel_Handler(The_Timer);
      Next_Start := Next_Start + Real_Time.To_Time_Span(Period);
      delay until Next_Start;
    end loop;
  end Periodic_Stopped;

...
```

Example 2: Lowered task

In this example, when an execution time overrun is detected the priority of the task is lowered to allow other lower priority tasks to continue meeting their timing 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.

```
with Ada.Real_Time,Ada.Execution_Time.Timers,

  Ada.Dynamic_Priorities, System, Ada.Task_Identification;
use Ada;

...

  protected Control is
    entry Wait_Budget_Expiry;
    procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer);
```

```
    pragma Priority(Ada.Execution_Time.Timers.Min_Handler_Ceiling);
  private
    Expired : Boolean := False;
  end Control;

  protected body Control is
    entry Wait_Budget_Expiry when Expired is
    begin
      Expired := False;
    end Wait_Budget_Expiry;

    procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer) is
    begin
      Expired := True;
    end Budget_Expired;
  end Control;

 task Worker;

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

     task body Supervisor is
       WCET : constant Duration:=1.0E-3;
       My_Id : aliased Task_Identification.Task_Id:=Supervisor'Identity;
       The_Timer : Ada.Execution_Time.Timers.Timer(My_Id'Access);
       Low_Prio : System.Priority:=System.Priority'First;
     begin
       Ada.Execution_Time.Timers.Set_Handler
         (The_Timer, Real_Time.To_Time_Span(WCET),
          Control.Budget_Expired'Access);
       Control.Wait_Budget_Expiry; -- wait for timer overrun
       Dynamic_Priorities.Set_Priority(Low_Prio,Worker'Identity);
     end Supervisor;
  begin
     -- Do_useful_work;
     abort Supervisor;
  end Worker;

...
```

!corrigendum D.14(01)

@dinsc

This clause describes a language-defined package to measure execution time.

@i<@s8<Static Semantics>>

The following language-defined library package exists:

@xcode<@b<with> Ada.Task_Identification;
@b<with> Ada.Real_Time; @b<use> Ada.Real_Time;
@b<package> Ada.Execution_Time @b<is>

```
   @b<type> CPU_Time @b<is private>;
   CPU_Time_First : @b<constant> CPU_Time;
   CPU_Time_Last  : @b<constant> CPU_Time;
   CPU_Time_Unit  : @b<constant> := @ft<@i<implementation-defined-real-number>>;
   CPU_Tick : @b<constant> Time_Span;

   @b<function> Clock
     (T : Ada.Task_Identification.Task_Id
         := Ada.Task_Identification.Current_Task)
    @b<return> CPU_Time;

   @b<function> "+"  (Left : CPU_Time; Right : Time_Span) @b<return> CPU_Time;
   @b<function> "+"  (Left : Time_Span; Right : CPU_Time) @b<return> CPU_Time;
   @b<function> "-"  (Left : CPU_Time; Right : Time_Span) @b<return> CPU_Time;
   @b<function> "-"  (Left : CPU_Time; Right : CPU_Time)  @b<return> Time_Span;

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

   @b<procedure> Split
     (T : @b<in> CPU_Time; SC : @b<out> Seconds_Count; TS : @b<out> Time_Span);

   @b<function> Time_Of (SC : Seconds_Count; TS : Time_Span) @b<return> CPU_Time;

@b<private>
   ... --  not specified by the language
@b<end> Ada.Execution_Time;>
```

The @i<execution time> or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. It is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers and run-time services on behalf of the system.

The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers.

The CPU_Time value I represents the half-open execution-time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation-defined real number. For each task, the execution time value is set to zero at some unspecified point between the creation of the task and the start of the activation of the task.

CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

@i<@s8<Dynamic Semantics>>

CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A @i<CPU clock tick> is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals.

The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types.

The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 + TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit.

@i<@s8<Erroneous Execution>>

For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous.

@i<@s8<Implementation Requirements>>

The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task start-up to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

@i<@s8<Documentation Requirements>>

The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

The implementation shall document the properties of the underlyingmechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

@i<@s8<Metrics>>

The implementation shall document the following metrics:

@xbullet<An upper bound on the execution-time duration of a clock tick. This is a value D such that if t1 and t2 are any execution times of a given task such that t1<t2 and Clock[t1]=Clock[t2] then t2-t1 <= D.>

@xbullet<An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution -time clock (as observed by calling the Clock function with the same Task_Id).>

@xbullet<An upper bound on the execution time of a call to the Clock function, in processor clock cycles.>

@xbullet<Upper bounds on the execution times of the operators of the type CPU_Time, in processor clock cycles.>

@i<@s8<Implementation Permissions>>

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the CPU_Time type.

@i<@s8<Implementation Advice>>

When appropriate, implementations should provide configuration mechanisms to change the value of CPU_Tick.

!corrigendum D.14.1(01)

@dinsc

This clause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time.

@i<@s8<Static Semantics>>

The following language-defined library package exists:

```
@b<with> System;
@b<package> Ada.Execution_Time.Timers @b<is>

   @b<type> Timer (T : @b<access> Ada.Task_Identification.Task_Id) @b<is>
     @b<limited private>;

   @b<type> Timer_Handler @b<is>
     @b<access protected procedure> (TM : @b<in out> Timer);

   Min_Handler_Ceiling : @b<constant> System.Any_Priority :=
    @ft<@i<implementation-defined>>;

   @b<procedure> Set_Handler (TM      : @b<in out> Timer;
                              In_Time : @b<in> Time_Span;
                              Handler : @b<in> Timer_Handler);
   @b<procedure> Set_Handler (TM      : @b<in out> Timer;
                              At_Time : @b<in> CPU_Time;
                              Handler : @b<in> Timer_Handler);
   @b<function> Current_Handler (TM : Timer) @b<return> Timer_Handler;
   @b<procedure> Cancel_Handler (TM        : @b<in out> Timer;
                                 Cancelled : @b<in out> Boolean);

   @b<function> Time_Remaining (TM : Timer) @b<return> Time_Span;

   Timer_Resource_Error : @b<exception>;

@b<private>
   ... --  not specified by the language
@b<end> Ada.Execution_Time.Timers;>
```

The type Timer represents an execution-time event for a single task and is capable of detecting execution time overruns. The access discriminant T identifies the task concerned. The type Timer needs finalization (see 7.6).

An object of type Timer is said to be @i<set> if it is associated with a non-null value of type Timer_Handler and @i<cleared> otherwise. All Timer objects are initially cleared.

The type Timer_Handler identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a
@i<handler>.

@i<@s8<Dynamic Semantics>>

When a Timer object is created, or upon the first call of a Set_Handler procedure with the timer as parameter, the resources required to operate a execution-time timer based on the associated execution-time clock are allocated and initialized. If this operation would exceed the available resources, Timer_Resource_Error is raised.

The procedures Set_Handler associate the handler Handler with the timer TM; if
Handler is @b<null>, the timer is cleared, otherwise it is set. The first procedure Set_Handler loads the timer TM with an interval specified by the Time_Span parameter. In this mode, when the execution time of the task identified by TM.T has increased by In_Time, the timer TM is said to have expired. The second procedure Set_Handler loads the timer TM with the absolute value specified by At_Time. In this mode, when the the execution time of the task identified by TM.T reaches At_Time, the timer TM is said to have @i<expired>; if the value of At_Time has already been reached when Set_Handler is called, the timer TM is said to be expired.

A call of a procedure Set_Handler for a timer that is already set replaces the handler and the (absolute or relative) execution time; if Handler is not
@b<null>, the timer remains set.

When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

The function Current_Handler returns the handler associated with the timer
TM if that timer is set; otherwise it returns @b<null>.

The procedure Cancel_Handler clears the timer if it is set. Cancelled is assigned True if the timer was set prior to it being cleared; otherwise it is assigned False.

The function Time_Remaining returns the execution time interval that remains until the timer TM would expire, if that timer is set; otherwise it returns
Time_Span_Zero.

The constant Min_Handler_Ceiling is the priority value that ensures that no ceiling violation would occur, were a handler to be executed.

For all the subprograms defined in this package, Tasking_Error is raised if the task identified by TM.T has terminated, and Program_Error is raised if the value of TM.T is Task_Identification.Null_Task_Id.

An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

@i<@s8<Erroneous Execution>>

For a call of any of the subprograms defined in this package, if the task identified by TM.T no longer exists, the execution of the program is erroneous.

@i<@s8<Implementation Requirements>>

For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

When an object of type Timer is finalized, the system resources used by the timer shall be deallocated.

@i<@s8<Implementation Permissions>>

Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded then Timer_Resource_Error is raised.

@xindent<@s9<NOTES@hr
A Timer_Handler can be associated with several Timer objects.>>

!ACATS test

ACATS test(s) should be created for this package.

!appendix

Additional references for the original proposal:

[1] A Proposal to Integrate the POSIX Execution-Time Clocks into Ada 95. Technical Report, June, 2002, By J. Miranda Gonzalez and M. Gonzalez Harbour.

[2] Implementing and using Execution Time Clocks in Ada Hard Real-Time Applications. By Gonzalez Harbour M., Aldea Rivas M., Gutierrez Garcia J.J., Palencia Gutierrez J.C. International Conference on Reliable Software Technologies, Ada-Europe'98, Uppsala, Sweden, in Lecture Notes on Computer Science No. 1411, Junio, 1998, ISBN:3-540-64563-5, pp. 91,101.

[3] Extending Ada's Real-Time Systems Annex with the POSIX Scheduling Services. By: Mario Aldea Rivas and Michael Gonzalez Harbour. IRTAW-2000, Las Navas, Avila, Spain.