User-Defined Clocks
Is it the right time now?

A.J. Wellings and A. Burns
Department of Computer Science, University of York
Heslington, York YO10 5DD, UK
(andy,burns)@cs.york.ac.uk

Abstract

During the Ada 9X design processes, user-defined time types and clocks were proposed. These, however, were dropped when the scope of proposed changes was reduced. This paper reviews the original proposal and suggests a new solution which is in keeping with Ada 2005. Our goal is to regenerate interest in the topic to see if it is worthy of support in a future amendment to Ada.

1 Introduction

The International Real-Time Ada Workshop series has a long history and has seem many changes to the real-time programming model of Ada. The biggest of these changes occurred going from Ada 83 to Ada 95. Many issues were discussed during this time and some ideas were dropped because of the total number of changes being requested were considered too large. One such issue was that of user-defined clocks and time types.

This paper argues that user-define clocks and time types are still important and reconsiders the issue in the light of the current language. Throughout our discussions, we assume the following definitions [2].

- A clock has **monotonicity**, if each successive time reading from that clock yields a time that is not before a previous reading.
- Two clocks are **synchronized** when the difference between their time values is less than some specified offset. Synchronization in general degrades with time, and may be lost, given a specified offset. The error is called **clock drift**.
- **Resolution** is the minimal time value interval that can be distinguished by a clock.
- **Uniformity** refers to the measurement of the progress of time at a consistent rate, with a tolerance on the variability. Uniformity is affected by two other factors, **jitter** and **stability**. Jitter is a short-term and non-cumulative small variation caused by noise sources, like thermal noise. More practically, jitter refers to the distribution of the differences between when events are actually fired or noticed by the software and when they should have really occurred according to time in the real-world. (Lack-of) stability accounts for large and often cumulative variations, due to e.g. supply voltage and temperature. In practice a clock is driven by an oscillator. Accuracy is the difference between the desired frequency and the actual frequency of the oscillator, and a major reason of synchronization loss.

2 Clocks and Time Types in Ada

Ada has the notion of a **time base** although there is no formal definition of what this means. For every time base there is an associated clock. The value of the time read from the clock is some transformation from its associated time base. Time values read from clocks are always private types. The type *duration* is not specific to a particular time base, but can be used with any time base (hence it is defined in package `Standard`).

In Ada 2005 there are three "time" types explicitly supported by the language

- `Ada.Calendar.Time` – This is provided by the main language and represents a "wall clock's" time. That is, one that recognised leap years, leap seconds and other adjustments. Relative time values are defined by the `Duration` type.

- `Ada.Real_Time.Time` – This is provided by the Real-Time Annex and is supported by a monotonic clock. Relative time values are represented by the `Time_Span` type. Conversions between the `Duration` and `Time_Span` type are supported.

- `Ada.Execution_Time.CPU_Time` – This is provided by the Real-Time Annex and is supported by a clock per task. A CPU-time clock only ticks when its associated task is executing. Relative time values are represented by the `Ada.Real_Time.Time_Span` type.

It should be noted that the language also allows other implementation-defined time types to be supported.

The common operations/properties of all the supported time types are:

- function Clock – returns the current time. `Ada.Calendar.Clock` is not monotonic, there is no required synchronization with any external time reference (such as UTC), and no requirement on uniformity. `Ada.Real_Time.Clock` is monotonic, there is no required synchronization with any external time reference (although an implementation should document the upper bound on the drift rate of Clock with respect to "real time") or with `Ada.Calendar.Clock`, but it must progress at a uniform rate. However, is recommended that both `Ada.Calendar.Clock` and `Ada.Real_Time.Clock` are transformation of the same underlying time base. `Ada.Execution_Time.Clock` is monotonic, there is no required synchronization with any other clock, and it does not have to progress at a uniform rate.

- Binary operators – to allow comparison of time objects and the addition/subtraction of relative time values to/from absolute time values etc.

- The notion of a tick. This is the maximum amount of time that the value from clock will remain unchanged. For `Calendar` time, it is the value of `System.Tick`. For `Real_Time` time it is given by `Ada.Real_Time.Tick`. For CPU time, it is given by `Ada.Execution_Time.CPU_Tick`.

There are some differences in the support for the three time types. For example

- The notion of an epoch – only the Real-Time clock has the notion of an epoch. All real-time absolute time values are measured as relative times from an "unspecified" epoch. The other time types do not explicitly mention an epoch, however it is defined that the CPU time of a newly created task is zero, and its CPU-time clock starts when the task begin its activation.

- Support for delays – the language supports both both relative delays (the *delay* statement) and absolute delays (the *delay_until* statement). However, only the time values of type duration can be used by a relative delay. The language requires that the time base for *delay_relative_statement*s should be monotonic, and it need not be the same time base as used for `Calendar.Clock`. **Given that the type duration can be associated with any time base, it is not clear how you can specify which time base should be used in *delay_relative_statement*.** Absolute delays are supports for both `Calendar.Time` and `Real_Time.Time` and any implementation-defined time type. It is not supported for CPU times.

- Support for Timers – timers are explicitly supported for real-time time values (via `Ada.Real_Time.Timing_Events`) and for CPU time values (via `Ada.Execution_Time.Timers`). Essentially they allow the application to be notified when either an absolute time has arrived, or a relative time has passed. Of course, the use of the *delay* and the *delay_until* statements imply the use of a timer.

3 User-Defined Clocks in Ada 9X

At the Fifth IRTAW held in Florida in March 1991, a draft Ada 9x mapping document was considered. One of the issues raised was that of "Further Extensions to Delays". The following is the verbatim write-up of this topic by Ted Baker in his Rapporteur's report[1]

The set of types for which delays are permitted would be extensible, by the language implementor or a skilled application programmer. The person making such an extension would need to provide a timer-device driver, and provide the compiler with information about the identity of the new time type and procedures to which calls should be generated by the compiler for the translation of delays on that type. The specific syntax for providing this information to the compiler had not yet been completely worked out. The fragmentary examples in the draft mapping document hinted at one syntax, and a variation was proposed on handwritten overhead projector transparencies during the discussions. The syntax in the examples below is a mixture of these two.

The other main issue is how to ensure the new time type satisfies some minimal requirements necessary for interaction with the compiler. The draft mapping proposed to ensure this by requiring that the type be derived from some pre-defined time type. The delay extension proposal is based on the proposed protected record mechanism, which was discussed in another session of the workshop. The Ada language implementation would be required to provide a predefined protected record type, say SIMPLE_PR, something like

```
protected type SIMPLE_PR is
  entry WAIT;
  procedure SIGNAL;
record
  OCCURRED : BOOLEAN := FALSE;
end SIMPLE_PR;
```

If PR were an object of type SIMPLE_PR, the operation PR.WAIT would cause the calling task to be suspended until PR.SIGNAL is called. The implementor of a timer extension would provide an interrupt handler for timer interrupts (real or simulated), which could wake delayed tasks up by calling the SIGNAL, operation on an object of type SIMPLE_PR associated with each delayed task. The association of this SIMPLE_PR object with the task is the responsibility of QUEUE and DEQUEUE operations which would be defined by the implementor of the extension. For each user-defined timer extension, the person implementing the extension would write a package something like the following

```
with INTERRUPT_MANAGEMENT;
package MY_CLOCK is
  -- written by the extender
  type MY_TIME is private;
  function CLOCK return MY_TIME;
private
  type MY_TIME is . . .
    -- e.g. 64-bit fixed point
  type DQEL;
  type DQ_LINK is access DQEL;
  type DQEL is
    record
      PR : SIMPLE PR;
      -- predefined PR type
      WAKEUP_TIME : MY_TIME;
      -- defined by the user
      NEXT : DQ_LINK;
      -- defined by the user
   end record;

  protected MY_TIMER is
    procedure HANDLE_INTERRUPT;
    pragma ATTACH INTERRUPT HANDLER
      (HANDLE_INTERRUPT'ACCESS,
      INTERRUPT_MANAGEMENT.INTERRUPT_NAMES.
      SPECIAL_TIMER_ID);
    procedure SET_ALARM(WAKEUP_TIME: MY_TIME);
    procedure ENQUEUE(DQEL : in out DQEL;
                      WAKEUP_TIME: MY_TIME);
    procedure DEQUEUE(DQEL: in out DQEL);
    function CLOCK return MY_TIME;
  record
    NEXT _ALARM : HY TINE := 0.0;
    CURRENT_TIME : MY TINE := 0.0;
    Q_HEAD : DQ_LINK := null;
  end HY TIMER;

  function CLOCK return MY_TIME
            renames MY_TIMER.CLOCK;
  procedure DQEL'ENQUEUE(DQEL: in out DQEL;
                         WAKEUP TIME: MY_TIME)
            renames MY_TIMER.ENQUEUE;
  procedure DQEL'DEQUEUE(DQEL: in out DQEL)
           renames MY TIMER.DEQUEUE;
  for MY_TIME'QUEUE ELEMENT use DQEL;
end MY_CLOCK;
```

The renamings and representation clause here are indicative of some standard mechanism, to be defined, by which the application would make the DQEL, type and the ENQUEUE and DEQUEUE operations known to the Ada implementation, for use in DELAY statements. If DQEL, is derived from a predefined delay queue element type, the operations ENQUEUE and DEQUEUE are predefined, but are overridden by the user. QUEUE_ELEMENT is a new language-defined attribute of time types. The requirement that DQEL: have the appropriate SIMPL_PR component would be enforced by the compiler.

The body of this package would contain code to wake up the delayed tasks when the clock defined by this package says it is time.

```
package body MY_CLOCK is
  protected body MY_TIMER is
    procedure HANDLE INTERRUPT is
      -- Invoked by the interrupt to
      -- wake up delayed tasks.
      TEMP : DQ_LINK;
    begin
      CURRENT_TIME :=
         READ_OR_CALCULATE_VALUE_FROM_DEVICE;
      while CURRENT_TIME >= Q_HEAD.WAKEUP_TIME
      loop
        TEMP := Q_HEAD;
        Q_HEAD := Q_HEAD.NEXT;
        TEMP.PR.SIGNAL;
           -- Awaken a delayed task.
      end loop;
      NEXT_ALARM := Q_HEAD.WAKEUP_TIME;
      SET_DEVICE_TO_INTERRUPT(NEXT_ALARM);
    end HANDLE_INTERRUPT;

    procedure SET_ALARM (
        WAKEUP_TIME : MY_TIME) is
    begin
      if WAKEUP_TIME < NEXT_ALARM then
        NEXT_ALARM := WAKEUP_TIME;
        SET_DEVICE_TO_INTERRUPT(NEXT_ALARM);
      end if;
    end SET ALARM;
    -- other subprogram bodies
  end MY_TIMER;
end MY_CLOCK;
```

When the compiler encounters a DELAY statement on an expression of a user-defined type, it uses the operations of the type to translate the statement according to a standard paradigm. For example, suppose *the following statement appeared in a program*

```
delay MY_CLOCK.MY_TIME'(D);
```

The compiler would be responsible for translating this into machine code equivalent to the following DECLARE block:

```
declare
  DQ : DQEL;
begin
  DQEL'ENQUEUE(DQ,D);
  DQ.PR.WAIT;
  DQEL'DEQUEUE(DQ);
end;
```

Similar (but more complex) schemes would be followed for occurrences of DELAY inside SELECT statements.

This extension mechanism allows Ada implementors and real-time application developers to provide multiple clocks. It would go toward satisfying the need for virtual clocks in simulation, and the need for a per-task clock that measures the time each task has used the processor.

Whilst some of the fine details of the approach are missing, the above discussion does illustrate the basic idea. One of its disadvantages is that the provider of the user-defined clock must also provide the management of the delay queues. This seems unnecessary, as the compiler should be able to provide this automatically. Such an approach is considered in the next section.

4 Proposal for Discussion

There are three main issues that are of interest.

- The definition of a "time base". The key issue is to be able to differentiate between time bases that can be used to support **active** and **passive** clocks. An active clock allows the association of software timers and hence time values supported by such a clock can also be used to support the *delay* and *delay_until* statements. The underlying time base for an active clock can be as simple as a hardware timer chip.

  A passive clock simply allows the current time to be read. It does not support timers, and time values supported by such a clock cannot be used in the *delay* and *delay_until* statements. An example of an underlying time base for such a clock is a CPU cycle counter or one that takes time from a GPS signal.

- The relation between the absolute and relative time supported by a clock and "real time". The key issue here is whether the epoch of a clock can be (directly) determined in relation to wall clock time. If it can, then does this imply that all time values from a clock can be mapped to time values from wall clock time. Also whether the type duration is appropriate for all clocks. For example, consider a time base that is provided by the rotation of a crankshaft. The full rotation of which represents the tick of the associated clock. This will depend on the speed of rotation and therefore absolute time vales will not have a direct correlation with wall clock time and the type `Duration` is not a relevant measure of relative time (although we note type `Duration` is a fixed point type and is only interpreted to mean seconds).

In order to generalize Ada's support for user-defined time types, we introduce a new package `Ada.General_Time`. Two types are introduced: `Time_Root` is the root type for all absolute time values and `Time_Interval_Root` is the root type for all relative time values.

```
with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
package Ada.General_Time is
  type Time_Root is abstract tagged null record;
  type Time_Interval_Root is
       abstract tagged null record;
  function Clock return Time_Root is abstract;
  function Tick return Time_Interval_Root is
           abstract;

  function "<"(Left : Time_Root;
               Right : Time_Root)
                return Boolean is abstract;
  function "+"(Left : Time_Root;
               Right : Time_Interval_Root'Class)
               return Time_Root is abstract;
  -- other operators similarly defined

  type Alarm_Handler_Absolute is access
        protected procedure (
        T: in Time_Root'Class);
  type Alarm_Handler_Relative is
         access protected procedure (
         T: in Time_Interval_Root'Class);

  procedure Set_Next_Alarm_At(
       T : in Time_Root;
       H : Alarm_Handler_Absolute)
       is abstract;
  procedure Set_Next_Alarm_After(
       T : in Time_Interval_Root;
       H : Alarm_Handler_Relative)
       is abstract;

  Not_Supported : exception;
    -- raised by Set_Next_Alarm_At and
    -- Set_Next_Alarm_After if the underlying
    -- time base does not active clocks

end Ada.General_Time;
```

Note that the assumption is that the procedures that set alarms only keeps track of the nearest alarm. The Ada run-time system will keep track of all alarms. However, it is an open issue whether a user-defined clock needs to be able to call-back to the Ada run-time if the absolute time has changed. For example, if the clock jumps forward, then it could be that an outstanding absolute delay now might occur before and previously nearer relative relay.

Note that the `Split` operation and operations that convert between the `Duration` and `Time_Interval_Root` types are not present at this level and must be added (if appropriate) by the user-defined time types.

The current definition of the *delay* and *delay until* statements given by the language remains unchanged:

```
delay_statement ::=
      delay_until_statement |
      delay_relative_statement

delay_until_statement ::=
      delay until delay_expression;
delay_relative_statement ::=
      delay delay_expression;
```

However, the name resolution and legality rules need to be extended.

> The Name Resolution Rules are that the expected type for the `delay_expression` in a delay_relative_statement is the predefined type `Duration` or a sub type of `Time_Interval_Root`. The `delay_expression` in a `delay_until_statement` is expected to be of any nonlimited type.
>
> The Legality Rules are that can be multiple time bases, each with a corresponding clock, and a corresponding time type. The type of the `delay_expression` in a `delay_until_statement` shall be a time type – either the type `Time` defined in the language-defined package `Calendar`, or a subtype of `Time_Root` or the type of some other implementation-defined time type.

4.1 Examples of user-defined time type

In this section we illustrate two simple use-defined clocks. An active clock that is tick-driven and can support timers. The other, a passive clock, that reads a GPS signal.

A tick-driven clock

In this example, the clock simply keeps track of the number of clock ticks since the system was booted. The underlying time base is a simple oscillator.

```
with Ada.General_Time; use Ada.General_Time;
with Calendar;
package My_Clock is
   type Time is new Time_Root with private;
   type Time_Interval is new Time_Interval_Root
        with private;

   overriding function Clock return Time;
   overriding function Tick return
        Time_Interval;

   overriding function "<"(
        Left : Time;
        Right : Time) return Boolean;
   overriding function "+"(
        Left : Time;
        Right : Time_Interval_Root'Class)
        return Time;

   overriding procedure Set_Next_Alarm_At(
        T : in Time;
        H : Alarm_Handler_Absolute);
   overriding procedure Set_Next_Alarm_After(
        T : in Time_Interval;
        H : Alarm_Handler_Relative);

   function Epoch return Calendar.Time;
   function To_Duration(Ti : Time_Interval)
        return Duration;
   function To_Time_Interval(D : Duration)
        return Time_Interval;
   -- other operations such as Split etc
private
   type Time is new Time_Root with
      record
         Ticks : Long_Integer;
      end record;

   type Time_Interval is new
        Time_Interval_Root with
      record
         Interval : Long_Integer;
      end record;

   protected Clock_Interrupt is
      pragma Priority(Interrupt_Priority'First);
      procedure Handler;
      pragma Interrupt_Handler(Handler);
      procedure Set_Next_Alarm_At(
          T : in Time;
          H : Alarm_Handler_Absolute);
      procedure Set_Next_Alarm_After(
          T : in Time_Interval;
          H : Alarm_Handler_Relative);
   private
      Next_Alarm : Long_Integer := 0;
      Next_T : Time;
      Next_I : Time_Interval;
      AH : Alarm_Handler_Absolute;
      IH : Alarm_Handler_Relative;
      Absolute : Boolean;
   end Clock_Interrupt;

   overriding procedure Set_Next_Alarm_At(
       T : in Time;
       H : Alarm_Handler_Absolute) renames
       Clock_Interrupt.Set_Next_Alarm_At;

   overriding procedure Set_Next_Alarm_After(
       T : in Time_Interval;
       H : Alarm_Handler_Relative) renames
     Clock_Interrupt.Set_Next_Alarm_After;
end My_Clock;
```

A protected object in the private part of the package provides the interface with the clock interrupt.

```
with Ada.Interrupts; use Ada.Interrupts;
with Ada.Interrupts.Names;
use Ada.Interrupts.Names;
package body My_Clock is
  Ticks_Since_Epoch : Long_Integer := 0;
  pragma atomic(Ticks_Since_Epoch);
  Epoch_Calendar_Time : Calendar.Time;

  Clock_Tick : constant Long_Integer := ...;
  function Clock return Time is
  begin
    return (Ticks => Ticks_Since_Epoch);
  end Clock;

  function Tick return Time_Interval is
  begin
    return (Interval => Clock_Tick);
  end;

  function "<"(Left : Time; Right : Time)
               return Boolean is
  begin
    return Left.Ticks < Right.Ticks;
  end "<";

  function "+"(Left : Time;
      Right : Time_Interval_Root'Class)
      return Time is
    Tmp : Time_Interval :=
          Time_Interval(Right);
  begin
    return (Ticks => Left.Ticks +
            Tmp.Interval);
  end "+";

  function Epoch return Calendar.Time is
  begin
    return Epoch_Calendar_Time;
  end Epoch;

  ...

  protected body Clock_Interrupt is
    procedure Handler is
    begin
      Ticks_Since_Epoch :=
            Ticks_Since_Epoch + 1;
      if Ticks_Since_Epoch = Next_Alarm then
        Next_Alarm := 0;
        if Absolute then
          AH(Next_T);
        else
          IH(Next_I);
        end if;
      end if;
    end Handler;

    procedure Set_Next_Alarm_At(
      T : in Time;
      H : Alarm_Handler_Absolute) is
    begin
      if T.Ticks <= Ticks_Since_Epoch then
        H(T);
      else
        if T.Ticks < Next_Alarm then
          Next_Alarm := T.Ticks;
          Absolute := True;
          AH := H;
          Next_T := T;
        end if;
      end if;
    end Set_Next_Alarm_At;

    procedure Set_Next_Alarm_After(
        T : in Time_Interval;
        H : Alarm_Handler_Relative) is
    begin
      if Ticks_Since_Epoch + T.Interval <
           Next_Alarm then
        Next_Alarm := Ticks_Since_Epoch +
                      T.Interval;
        Absolute := False;
        IH := H;
        Next_I := T;
      end if;
    end;
  end Clock_Interrupt;
begin
   Attach_Handler(Clock_Interrupt.Handler'Access,
                  On_Board_Clock);
   Epoch_Calendar_Time := Calendar.Clock;
end My_Clock;
```

A GPS-based clock

The Global Positioning System (GPS) also provides a source of “global time” and hence can be used as a time base for a passive clock. Here, we assume that a GPS device is accessed by a serial line. To get the time, a command must be sent to retrieve GPS data that contains the time the data was read. This can then be extracted. An interrupt in the serial line indicates that the data is available.

The package specification is given below. For simplicity, the internal representation of `Time` is the same as `Calendar.Time`.

```
with Ada.Calendar; with Ada_General_time;
use Ada_General_Time;
package GPS_Clock is

  type Time is new Time_Root with private;
  type Time_Interval is new Time_Interval_Root
       with private;
  overriding function Clock return Time;
  overriding function Tick return Time_Interval;

  overriding function "<"(Left : Time;
      Right : Time) return Boolean;
  overriding function "+"(Left : Time;
      Right : Time_Interval_Root'Class)
      return Time;

  overriding procedure Set_Next_Alarm_At(
      T : in Time;
      H : Alarm_Handler_Absolute);

  overriding procedure Set_Next_Alarm_After(
      T : in Time_Interval;
      H : Alarm_Handler_Relative);
  function Epoch return Ada.Calendar.Time;
  function To_Duration(Ti : Time_Interval)
      return Duration;
  function To_Time_Interval(D : Duration)
      return Time_Interval;
  -- declarations for Split, Time_Of  etc
private
  type Time is new Time_Root with
    record
      ToD : Ada.Calendar.Time;
    end record;

  type Time_Interval is new
       Time_Interval_Root with
    record
      Interval : Duration;
    end record;
end GPS_Clock;
```

The body of the package simply contains the interface to the serial line.

```
with System; use System;
package body GPS_Clock is

  protected Serial_Line_Interrupt is
    pragma Priority(Interrupt_Priority'First);
    procedure Handler;
    pragma Interrupt_Handler(Handler);
    entry Wait_Interrupt;
  private
  Done : Boolean := False;
  end Serial_Line_Interrupt;

  protected body Serial_Line_Interrupt is separate;
    -- not given

  function Clock return Time is
    T: Time;
  begin
    -- send command to serial line to
    -- request GPS reading
    Serial_Line_Interrupt.Wait_Interrupt;
    -- read data back fro GPS and place in T
    return (T);
  end Clock;

  function Tick return Time_Interval is
  begin
    return (Interval => 0.001);
  end Tick;


  function "<"(Left : Time; Right : Time)
               return Boolean is separate;
     -- not given



  function "+"(Left : Time;
               Right : Time_Interval_Root'Class)
               return Time is separate;
     -- not given


  procedure Set_Next_Alarm_At(T : in Time;
            H : Alarm_Handler_Absolute) is
  begin
    raise Not_Supported;
  end Set_Next_Alarm_At;

  procedure Set_Next_Alarm_After(T : in Time_Interval;
            H : Alarm_Handler_Relative) is
  begin
    raise Not_Supported;
  end Set_Next_Alarm_After;

  function Epoch return Ada.Calendar.Time is
  begin
    -- The GPS epoch is 0000 UT (midnight) on January 6, 1980.
    return Ada.Calendar.Time_Of(1980, 1, 6, 0.0);
  end Epoch;

  function To_Duration(Ti : Time_Interval) return Duration is
  begin
    return Ti.Interval;
  end To_Duration;

  function To_Time_Interval(D : Duration) return Time_Interval is
  begin
    return (Interval=> D);
  end To_Time_interval;

  -- implementation for Split, Time_Of  etc
begin
  -- attach interrupt handler
end GPS_Clock;
```

5 Conclusions

As embedded systems become more and more complex, so it is likely that different sources of time will be required (for example, global time, distributed time, simulation time). It is not practical for a language to anticipate all the requirements that will be placed on it in this area. Ada 2005 currently allows an implementation to define its own time types and make them available to the programmer via implementation-dependent packages. Here we propose that the language should support a root type for all time and interval types. Indeed, this was original proposed back during the early 90s for inclusion into Ada 95. However, for non technical reason the proposal was dropped.

Acknowledgements

The authors gratefully acknowledge discussion held in JSR 282 about the introduction of user-defined clocks into the RTSJ. In particular Kelvin Nilsen for re-stimulating our interest in this topic.

References

[1] T. Baker. Time-related issues in Ada 9X. In *Proceedings of IRTAW 5*, pages 8–14, 1991.

[2] P. Dibble, R. Belliardi, B. Brosgol, D. Holmes, and A.J. Wellings. *The Real-Time Specification for Java*. www.rtjs.org, 2008.