John Barnes

Ada Rationale 2012

Tasking and Real-Time

Courtesy of

AdaCore

Rationale for Ada 2012: 4 Tasking and Real-Time

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements in the tasking and real-time areas for Ada 2012.

The most important is perhaps the recognition of the need to provide control over task allocation on multiprocessor architectures.

There are also various improvements to the scheduling mechanisms and control of budgets with regard to interrupts.

An interesting addition to the core language is the ability to specify restrictions on how a procedure of a synchronized interface is to be implemented.

Keywords: rationale, Ada 2012.

1 Overview of changes

The WG9 guidance document [1] identifies real-time systems as an important application area for Ada. In particular it says that attention should be paid to

> improving the capabilities of Ada on multicore and multiprocessor architectures.

Ada 2012 does indeed address the issues of multiprocessors as well as other real-time improvements.

The following Ada Issues cover the relevant changes and are described in detail in this paper:

30	Requeue on synchronized interfaces
117	Memory barriers and Volatile objects
166	Yield for non-preemptive dispatching
167	Affinities for programs on multiprocessor platforms
168	Extended suspension objects
169	Group budgets for multiprocessors
170	Monitoring time spent in interrupt handlers
171	Pragma CPU and Ravenscar profile
174	Implement task barriers in Ada
215	Pragma Implemented should be an aspect
278	Set_CPU called during a protected action

These changes can be grouped as follows.

First there are a number of improvements and additions to the scheduling mechanisms (166, 168, 174). These are in the Real-Time Systems annex (D).

A number of additions recognise the importance of the widespread introduction of multiprocessors and provide mechanisms for associating tasks with particular CPUs or groups of CPUs known as dispatching domains (167, 171, 278). There is an associated change to group budgets which were introduced in Ada 2005 (169). These changes also concern Annex D.

Other changes concerning budgets relate to the time spent in interrupt handlers (170). In some systems it may be possible to account for time spent in individual interrupts whereas in others it might only be possible to account for time spent in interrupts as a whole. Again these changes concern Annex D.

The definition of Volatile is updated to take account of multiprocessors (117).

Finally, there are changes to the core language regarding synchronized interfaces and requeue (30, 215).

2 Scheduling

Ada 83 was remarkably silent about the scheduling of tasks. It muttered about tasks being implemented on multiprocessors or using interleaved execution on a single processor. But it said nothing about how such interleaving might be achieved. It also indicated that a single Ada task might be implemented using several actual processors if the effect would be the same.

Ada 83 introduced the pragma Priority and stated

> if two task with different priorities are both eligible for execution ... then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.

The Rationale for Ada 83 says that this rule requires preemptive scheduling. But it says nothing about what happens if several tasks have the same priority. It does however have a dire warning

> Priorities are provided as a tool for indicating relevant degrees of urgency and on no account should their manipulation be used as a technique for attempting to obtain mutual exclusion.

So, apart from the existence of priorities, implementations were free to use whatever scheduling algorithms they liked such as Round Robin time slicing or simply running until blocked.

There was also a bit of a mystery about the delay statement. On the one hand Ada 83 says

> suspends execution of the task for at least the duration specified.

The words "at least" caused much confusion. The intent was simply a reminder that a task might not get the processor back at the end of the interval because another task might have become eligible for execution meanwhile. It did not mean that the implementation could willy-nilly delay execution for a longer time.

Another mystery surrounded the meaning of

```
delay 0.0;
```

Ada 83 did state that delay with a negative value is equivalent to a delay statement with a zero value. But it did not say what a delay with a zero value meant. The Rationale remained mute on the topic as well.

However, a general convention seemed to arise that **delay** 0.0; indicated that the task was willing to relinquish the processor and so force a scheduling point.

Ada 95 brought some clarity to the situation in the new Real-Time Systems annex by introducing the pragma Task_Dispatching_Policy and the standard argument of FIFO_Within_Priorities. But the core language did not clarify the effect of a delay of zero. It does say that a delay causes a task to be blocked but if the expiration time has already passed, the task is not blocked. So clearly a negative delay does not block. However, it still has the note that a negative delay is equivalent to delay zero so we could deduce that delay zero does not block and so cannot force scheduling.

But help is at hand in the Real-Time Systems annex where it clearly states that even if a delay does not result in blocking, nevertheless the task goes to the end of the ready queue for its active priority. But that is only for the standard policy of FIFO_Within_Priorities. If a malevolent vendor introduces a curious policy called perhaps Dodgy_Scheduling then it need not follow this rule.

Ada 2005 added further policies namely

```
Non_Preemptive_FIFO_Within_Priorities

Round_Robin_Within_Priorities
```

```
EDF_Across_Priorities
```

In the case of Non_Preemptive_FIFO_Within_Priorities a non-blocking delay also sends the task to the end of the ready queue for its active priority. However, a non-blocking delay has absolutely no effect in the case of Round_Robin_Within_Priorities and EDF_Across_Priorities.

The introduction of non-preemptive dispatching revealed a shortcoming that is cured in Ada 2012. The problem is that in such a system there is a need to be able to indicate that a task is willing to be preempted by a task of a higher priority but not by one of the same priority. So somehow we need to say Yield_To_Higher.

Moreover, some felt that it was time to get rid of this strange habit of writing **delay** 0.0; to indicate a scheduling point. Those restricted to the Ravenscar profile, had been forced to write something really gruesome such as

```
delay until Ada.Real_Time.Time_First;
```

Accordingly, the procedure Yield is added to the package Ada.Dispatching so that it becomes

```
package Ada.Dispatching is
  pragma Preelaborate(Dispatching);
  procedure Yield;
  Dispatching_Policy_Error: exception;
end Ada.Dispatching;
```

Calling Yield is exactly equivalent to **delay** 0.0; and similarly causes a bounded error if called from within a protected operation.

There is also a new child package thus

```
package Ada.Dispatching.Non_Preemptive is
  pragma Preelaborate(Non_Preemptive);
  procedure Yield_To_Higher;
  procedure Yield_To_Same_Or_Higher renames Yield;
end Ada.Dispatching.Non_Preemptive;
```

Calling Yield_To_Higher provides the additional facility required for non-preemptive scheduling. Note that, unlike Yield, it can be called from within a protected operation and does not cause a bounded error.

The pedantic programmer can call the precisely named Yield_To_Same_Or_Higher which simply renames Yield in the parent package.

Incidentally, note that since Yield has a side effect, Ada.Dispatching has been downgraded to preelaborable whereas it was pure in Ada 2005.

We now turn to consider an interaction between suspension objects introduced in Ada 95 and EDF scheduling introduced in Ada 2005.

Remember that suspension objects are manipulated by the following package

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S: in out Suspension_Object);
  procedure Set_False(S: in out Suspension_Object);
  function Current_State(S: Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S: in out Suspension_Object);
private
```

```
   ...
end Ada.Synchronous_Task_Control;
```

The state of a suspension object can be set by calls of Set_True and Set_False. The key feature is that the procedure Suspend_Until_True enables a task to be suspended until the suspension object is set true by some other task. Thus this provides a neat mechanism for signalling between tasks.

Earliest Deadline First (EDF) scheduling is manipulated by the following child package of Ada.Dispatching introduced in Ada 2005 (with use clauses added to save space)

```
with Ada.Real_Time; with Ada.Task_Identification;
use Ada.Real_Time; use Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline: constant Deadline := Time_Last;

  procedure Set_Deadline(D: in Deadline; TT: in Task_Id := Current_Task);
  procedure Delay_Until_And_Set_Deadline(Delay_Until_Time: in Time;
                                                     Deadline_Offset: in Time_Span);
  function Get_Deadline(T: Task_Id := Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

The procedure Delay_Until_And_Set_Deadline is the key feature. It enables a task to be blocked until the time given by the parameter Delay_Until_Time and sets the deadline so that it is Deadline_Offset after that.

But what is missing in Ada 2005 is the ability for a sporadic task triggered by a suspension object to have its deadline set in a similar manner. This is remedied in Ada 2012 by the addition of the following child package

```
with Ada.Real_Time;
package Ada.Synchronous_Task_Control.EDF is
  procedure Suspend_Until_True_And_Set_Deadline(S: in out Suspension_Object;
                                                              TS: in Ada.Real_Time.Span);
end Ada.Synchronous_Task_Control.EDF;
```

This enables a task to be blocked until the suspension object S is set true; it then becomes ready with a deadline of Ada.Real_Time.Clock + TS.

The other new feature concerning scheduling in Ada 2012 is the addition of a package Ada.Synchronous_Barriers. This enables many tasks to be blocked and to be released together.

The rationale for needing this facility is explained in the AI concerned. As general purpose computing is moving to parallel architectures and eventually to massively parallel machines, there is a need to efficiently schedule many tasks using barrier primitives. The POSIX OS interface provides a barrier primitive where *N* tasks wait on a barrier and are released simultaneously when all are ready to execute.

There are many situations where the release of *N* tasks is required to execute an algorithm in parallel. Often the calculation is relatively small for each task on each iteration but the number of tasks is relatively high. As an example consider the solution of partial differential equations where one task is allocated to each node of a grid; there might easily be several thousand nodes. Such an example is outlined in [2]. The cost of linearly scheduling and releasing them could remove almost all gains made through parallelization in the first place.

The new package is

```
package Ada.Synchronous_Barriers is
  pragma Preelaborate(Synchronous_Barriers);

  subtype Barrier_Limit is range 1 .. implementation-defined;
  type Synchronous_Barrier(Release_Threshold: Barrier_Limit) is limited private;
  procedure Wait_For_Release(The_Barrier: in out Synchronous_Barrier;
                                            Notified: out Boolean);
private
  ...
end Ada.Synchronous_Barriers;
```

The type Synchronous_Barrier has a discriminant whose value indicates the number of tasks to be waited for. When an object of the type is declared its internal counter is set to zero. Thus we might write

```
SB: Synchronous_Barrier(Release_Threshold => 100);
```

When a task calls the procedure Wait_For_Release thus

```
Wait_For_Release(SB, My_Flag);
```

then the task is blocked and the internal counter in SB is incremented. If the counter is then equal to the release threshold for that object (100 in this example), then all the tasks are released. Just one task will have the parameter Notified set to true (the mechanism for selecting the chosen task is not defined). This specially chosen task is then expected to do some work on behalf of all the others. Typically all the tasks will be of the same task type so the code of that type might have

```
Wait_For_Release(SB, My_Flag);
if My_Flag then                    -- Gosh, I am the chosen one
  ...                              -- do stuff
end if;
```

Once all the tasks are released, the counter in SB is reset to zero so that the synchronous barrier can be used again.

Care is needed regarding finalization, aborting tasks and other awkward activities. For example, if a synchronous barrier is finalized, then any tasks blocked on it are released and Program_Error is raised at the point of the call of Wait_For_Release.

Many embedded real-time programs, such as those conforming to the Ravenscar profile, run forever. However, there are soft multitasking programs which are hosted on systems such as Windows or Linux and these require closing down in an orderly manner. There are also programs that have mode changes in which the set of tasks involved can be changed dramatically. In such situations it is important that synchronous barriers are finalized neatly.

3 Multiprocessors

In recent years the cost of processors has fallen dramatically and for many applications it is now more sensible to use several individual processors rather than one high performance processor.

Moreover, society has got accustomed to the concept that computers keep on getting faster. This makes them applicable to more and more high volume but low quality applications. But this cannot go on. The finite value of the velocity of light means that increase in processor speed can only be achieved by using devices of ever smaller size. But here we run into problems concerning the nonzero size of Planck's constant. When devices get very small, quantum effects cause problems with reliability.

No doubt, in due course, genuine quantum processors will emerge based perhaps on attributes such as spin. But meanwhile, the current approach is to use multiprocessors to gain extra speed.

One special feature of Ada 2012 aimed at helping to use multiprocessors is the concept of synchronous barriers which were described above. We now turn to facilities for generally mapping tasks onto numbers of processors.

The key feature is a new child package of System thus

```
package System.Multiprocessors is
  pragma Preelaborate(Multiprocessors);

  type CPU_Range is range 0 .. implementation-defined;
  Not_A_Specific_CPU: constant CPU_Range := 0;
  subtype CPU is CPU_Range range 1 .. CPU_Range'Last;

  function Number_Of_CPUs return CPU;
end System.Multiprocessors;
```

Note that this is a child of System rather than a child of Ada. This is because System is generally used for hardware related features.

Processors are given a unique positive integer value from the subtype CPU. This is a subtype of CPU_Range which also includes zero; zero is reserved to mean not allocated or unknown and for clarity is the value of the constant Not_A_Specific_CPU.

The total number of CPUs is determined by calling the function Number_Of_CPUs. This is a function rather than a constant because there could be several partitions with a different number of CPUs on each partition. And moreover, the compiler might not know the number of CPUs anyway.

Since this is not a Remote Types package, it is not intended to be used across partitions. It follows that a CPU cannot be used by more than one partition. The allocation of CPU numbers to partitions is not defined; each partition could have a set starting at 1, but they might be numbered in some other way.

Tasks can be allocated to processors by an aspect specification. If we write

```
task My_Task
  with CPU => 10;
```

then My_Task will be executed by processor number 10. In the case of a task type then all tasks of that type will be executed by the given processor. The expression giving the processor for a task can be dynamic.

Moreover, in the case of a task type, the CPU can be given by a discriminant. So we can have

```
task type Slave(N: CPU_Range)
  with CPU => N;
```

and then we can declare

```
Tom: Slave(1);
Dick: Slave(2);
Harry: Slave(3);
```

and Tom, Dick and Harry are then assigned CPUs 1, 2 and 3 respectively. We could also have

```
Fred: Slave(0);
```

and Fred could then be executed by any CPU since 0 is Not_A_Specific_CPU.

The aspect can also be set by a corresponding pragma CPU. (This is an example of a pragma born obsolescent as explained in the paper on Contracts and Aspects.) The aspect CPU can also be given to the main subprogram in which case the expression must be static.

Further facilities are provided by the child package System.Multiprocessors.Dispatching_Domains as shown below. Again we have added use clauses to save space and also have often abbreviated Dispatching_Domain to D_D.

```
with Ada.Real_Time; with Ada.Task_Identification;
use Ada.Real_Time; use Ada.Task_Identification;
package System.Multiprocessors.Dispatching_Domains is
  pragma Preelaborate(Dispatching_Domains);

  Dispatching_Domain_Error: exception;

  type Dispatching_Domain(<>) is limited private;
  System_Dispatching_Domain: constant D_D;

  function Create(First, Last: CPU) return D_D;
  function Get_First_CPU(Domain: D_D) return CPU;
  function Get_Last_CPU(Domain: D_D) return CPU;
  function Get_Dispatching_Domain(T: Task_Id := Current_Task) return D_D;

  procedure Assign_Task(Domain: in out Dispatching_Domain;
                                    CPU: in CPU_Range := Not_A_Specific_CPU;
                                    T: in Task_Id := Current_Task);

  procedure Set_CPU(CPU: in CPU_Range; T: in Task_Id := Current_Task);

  function Get_CPU(T: in Task_Id := Current_Task) return CPU_Range;

  procedure Delay_Until_And_Set_CPU(Delay_Until_Time: in Time;
                                                    CPU: in CPU_Range);
private
  ...
end System.Multiprocessors.Dispatching_Domains;
```

The idea is that processors are grouped together into dispatching domains. A task may then be allocated to a domain and it will be executed on one of the processors of that domain.

Domains are of the type Dispatching_Domain. This has unknown discriminants and consequently uninitialized objects of the type cannot be declared. But such an object can be initialized by the function Create. So to declare My_Domain covering processors from 10 to 20 inclusive we can write

```
My_Domain: Dispatching_Domain := Create(10, 20);
```

All CPUs are initially in the System_Dispatching_Domain. A CPU can only be in one domain. If we attempt to do something silly such as create overlapping domains by for example also writing

```
My_Domain_2: Dispatching_Domain := Create(20, 30);
```

then Dispatching_Domain_Error is raised because in this case, CPU number 20 has been assigned to both My_Domain and My_Domain_2.

The environment task is always executed on a CPU in the System_Dispatching_Domain. Clearly we cannot move all the CPUs from the System_Dispatching_Domain other wise the environment task would be left high and dry. Again an attempt to do so would raise Dispatching_Domain_Error.

A very important rule is that Create cannot be called once the main subprogram is called. Moreover, there is no operation to remove a CPU from a domain once the domain has been created. So the general approach is to create all domains during library package elaboration. This then sets a fixed arrangement for the program as a whole and we can then call the main subprogram.

Each partition has its own scheduler and so its own set of CPUs, dispatching domains and so on.

Tasks can be assigned to a domain in two ways. One way is to use an aspect

```
task My_Task
  with Dispatching_Domain => My_Domain;
```

If we give both the domain and an explicit CPU thus

```
task My_Task
  with CPU => 10, Dispatching_Domain => My_Domain;
```

then they must be consistent. That is the CPU given must be in the domain given. If it is not then task activation fails (hands up all those readers who thought it was going to raise Dispatching_Domain_Error). If for some reason we write

```
task My_Task
  with CPU => 0, Dispatching_Domain => My_Domain;
```

then no harm is done. Remember that there is not a CPU with number zero but zero simply indicates Not_A_Specific_CPU. In such a case it would be better to write

```
task My_Task
  with CPU => Not_A_Specific_CPU, Dispatching_Domain => My_Domain;
```

The other way to assign a task to a domain is by calling the procedure Assign_Task. Thus the above examples could be written as

```
Assign_Task(My_Domain, 10, My_Task'Identity);
```

giving both domain and CPU, and

```
Assign_Task(My_Domain, T => My_Task'Identity);
```

which uses the default value Not_A_Specific_CPU for the CPU.

Similarly, we can assign a CPU to a task by

```
Set_CPU(A_CPU, My_Task'Identity);
```

Various checks are necessary. If the task has been assigned to a domain there is a check to ensure that the new CPU value is in that domain. If this check fails then Dispatching_Domain_Error is raised. Of course, if the new CPU value is zero, that is Not_A_Specific_CPU then it simply means that the task can then be executed on any CPU in the domain.

To summarize the various possibilities, a task can be assigned a domain and possibly a specific CPU in that domain. If no specific CPU is given then the scheduling algorithm is free to use any CPU in the domain for that task.

If a task is not assigned to a specific domain then it will execute in the domain of its activating task. In the case of a library task the activating task is the environment task and since this executes in the System_Dispatching_Domain, this will be the domain of the library task.

The domain and any specific CPU assigned to a task can be set at any time by calls of Assign_Task and Set_CPU. But note carefully that once a task is assigned to a domain other than the system dispatching domain then it cannot be assigned to a different domain. But the CPU within a domain can be changed at any time; from one specific value to another specific value or maybe to zero indicating no specific CPU.

It is also possible to change CPU but for the change to be delayed. Thus we might write

```
Delay_Until_And_Set_CPU(Delay_Until_Time => Sometime, CPU => A_CPU);
```

Recall we also have Delay_Until_And_Set_Deadline in Ada.Dispatching.EDF mentioned earlier.

Note that calls of Set_CPU and Assign_Task are defined to be task dispatching points. However, if the task is within a protected operation then the change is deferred until the next task dispatching point for the task concerned. If the task is the current task then the effect is immediate unless it is within a protected operation in which case it is deferred as just mentioned. Finally, if we pointlessly assign a task to the system dispatching domain when it is already in that domain, then nothing happens (it is not a dispatching point).

There are various functions for interrogating the situation regarding domains. Given a domain we can find its range of CPU values by calling the functions Get_First_CPU and Get_Last_CPU. Given a task we can find its domain and CPU by calling Get_Dispatching_Domain and Get_CPU. If a task is not assigned a specific CPU then Get_CPU naturally returns Not_A_Specific_CPU.

In order to accommodate interrupt handling the package Ada.Interrupts is slightly modified and now includes the following function

```
function Get_CPU(Interrupt: Interrupt_Id) return Systems.Multiprocessors.CPU_Range;
```

This function returns the CPU on which the handler for the given interrupt is executed. Again the returned value might be Not_A_Specific_CPU.

The Ravenscar profile is now defined to be permissible with multiprocessors. However, there is a restriction that tasks may not change CPU. Accordingly the definition of the profile now includes the following restriction

```
No_Dependence => System.Multiprocessors.Dispatching_Domains
```

In order to clarify the use of multiprocessors with group budgets the package Ada.Execution_Time.Group_Budgets introduced in Ada 2005 is slightly modified. The Ada 2005 version is

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

  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access protected procedure (GB: in out Group_Budget);

   ...                  -- and so on
private
  ...
end Ada.Execution_Time.Group_Budgets;
```

However, in Ada 2012 the type Group_Budget has a discriminant giving the CPU thus

```
type Group_Budget(CPU: System.Multiprocessors.CPU :=
                                        System.Multiprocessors.CPU'First)
                                                  is tagged limited private;
```

This means that a group budget only applies to a single processor. If a task in a group is executed on another processor then the budget is not consumed. Note that the default value for CPU is CPU'First which is always 1.

4 Interrupt timers and budgets

It will be recalled that Ada 2005 introduced three packages for monitoring the CPU time used by tasks. They are a root package Ada.Execution_Time plus two child packages thus

Ada.Execution_Time – this is the root package and enables the monitoring of execution time of individual tasks.

Ada.Execution_Time.Timers – this provides facilities for defining and enabling timers and for establishing a handler which is called by the run time system when the execution time of the task reaches a given value.

Ada.Execution_Time.Group_Budgets – this enables several tasks to share a budget and provides means whereby action can be taken when the budget expires.

The execution time of a task, or CPU time, is the time spent by the system executing the task and services on its behalf. CPU times are represented by the private type CPU_Time declared in the root package Ada.Execution_Time.

However, it was left implementation defined in Ada 2005 as to how the time spent in interrupts was to be accounted. The Ada 2005 RM says

> 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.

As noted in the AI, a common and simple implementation will charge the time consumed by the interrupt handlers to the task executing when the interrupt is generated. This is done under the assumption that the effect of interrupt handlers on the execution time clocks is negligible since the interrupt handlers are usually very short pieces of code. However, in real-time systems that undertake an intensive use of interrupts, this assumption may not be realistic. For example, Ada 2005 introduced timed events that can execute handlers in interrupt context. The facility is convenient and has low overheads, and therefore programmers are tempted to put more code into these handlers.

It is thus considered important to be able to measure time spent in interrupts and so facilities to do this are added in Ada 2012.

The root package is extended by the addition of two Boolean constants, Interrupt_Clocks_Supported and Separate_Interrupt_Clocks_Supported, and also a function Clocks_For_Interrupts so in outline it becomes

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

  type CPU_Time is private;

  ...

  function Clock(T: Task_Id := Current_Task) return CPU_Time;

  ...

  Interrupt_Clocks_Supported: constant Boolean := implementation-defined;
  Separate_Interrupt_Clocks_Supported: constant Boolean := implementation-defined;

  function Clocks_For_Interrupts return CPU_Time;

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

The constant Interrupt_Clocks_Supported indicates whether the time spent in interrupts is accounted for separately from the tasks and then Separate_Interrupt_Clocks_Supported indicates whether the time is accounted for each interrupt individually.

The new function Clocks_For_Interrupts returns the CPU_Time used over all interrupts. It is initialized to zero.

Time accounted for in interrupts is not also accounted for in individual tasks. In other words there is never any double accounting.

Calling the function Clocks_For_Interrupts if Interrupt_Clocks_Supported is false raises Program_Error. Note that the existing function Clock has a parameter giving the task concerned whereas Clocks_For_Interrupts does not since it covers all interrupts.

A new child package of Ada.Execution_Time is provided for monitoring the time spent in individual interrupts. Note that this package always exists even if the Boolean constant Separate_Interrupt_Clocks_Supported is false. Its specification is

```
package Ada.Execution_Time.Interrupts is
  function Clock(Interrupt: Ada.Interrupts.Interrupt_Id) return CPU_Time;
  function Supported(Interrupt: Ada.Interrupts.Interrupt_Id) return Boolean;
end Ada.Execution_Time.Interrupts;
```

The function Supported indicates whether the time for a particular interrupt is being monitored. If it is then Clock returns the accumulated CPU_Time spent in that interrupt handler (otherwise it returns zero). However, if the overall constant Separate_Interrupt_Clocks_Supported is false then calling this function Clock for any particular interrupt raises Program_Error.

The package Ada.Execution_Time.Timers is exactly the same in Ada 2012. However, as mentioned earlier, the package Ada.Execution_Time.Group_Budgets is now defined to work on a single processor and the type Group_Budget is modified to include a discriminant giving the CPU concerned.

5 Volatile

This is a curious topic and created much debate. For the collector of statistics the real part of the AI is less than two pages but the appendix has nearly twenty pages of chatter!

The problem is all about sharing variables and ensuring that things happen in the correct order. Moreover, we need to avoid the overhead of protected objects particularly on microprocessors where we might be using low level features such as memory barriers discussed in Section 2 above.

Suppose we have two tasks A and B which access some shared data perhaps in a nice package Common thus

```
package Common is
  ...
  Data: Integer;
  pragma Volatile(Data);
  Flag: Boolean;
  pragma Volatile(Flag);
  ...
end Common;
```

and in task A we write

```
with Common; use Common;
task A is
  ...
  Data := 42;
  Flag := True;
  ...
end A;
```

whereas in task B we have

```
with Common; use Common;
task B is
  Copy: Integer;
begin
  ...
  loop
    exit when Flag;                  -- spin
  end loop;
  Copy := Data;
  ...
end B;
```

The idea is that task A assigns some value to Data and then indicates this to task B by setting Flag to true. Meanwhile, task B loops checking Flag and when it is found to be true, then reads the Data.

Does this work in Ada 2005? Hmm. Nearly. There are three things that need to be ensured. One is that Flag gets changed in one lump. Another is that the new value of Data assigned by task A truly is updated when task B reads it. And the third is that the actions happen sequentially. Well, we should have applied pragma Atomic to Flag to ensure the first but since it is of type Boolean we might get away with it. And note that Atomic implies Volatile anyway. Also Atomic ensures that the actions are sequential.

So, with the pragma Volatile changed to Atomic for Flag, it does indeed work in Ada 2005 because Volatile ensures that read and writes are to memory and so things do happen in the correct order. However, this is overkill. It is not necessary that all accesses are to memory; all that matters is that they happen in the correct order so they could be to some intermediate cache. Indeed, there might be nested caches and as hardware evolves it is becoming more difficult to make general statements about its structure; hence we can really only make statements about the effect.

The possibility of introducing a new pragma Coherent was debated for some time. However, it was ultimately concluded that the definition of Volatile should be weakened. In Ada 2005 it says

> For a volatile object all reads and updates of the object as a whole are performed directly to memory.

In Ada 2012 it says

> All tasks of the program (on all processors) that read or write volatile variables see the same order of updates to the variables.

Of course, in Ada 2012, we use aspects so the package Common becomes

```
package Common is
  ...
  Data: Integer
    with Volatile;
  Flag: Boolean
    with Atomic;           -- Atomic implies Volatile
  ...
end Common;
```

where we have given Atomic for Flag. As mentioned above, Atomic implies Volatile so it is not necessary to give both. However, if we do have to give two aspects, it is much neater that the one aspect specification does this whereas two distinct pragmas would be necessary.

It is said that this change brings the meaning of volatile into line with that in C. However, it has also been said that the definition of volatile in C is unclear.

6 Synchronized interfaces and requeue

Ada 2005 introduced interfaces of various kinds: limited, nonlimited, synchronized, task, and protected. These form a hierarchy and in particular task and protected interfaces are forms of synchronized interfaces. The essence of this was to integrate the OO and real-time features of Ada. However, a problem was discovered regarding requeue as described in a paper presented at IRTAW 2007 [3].

Some examples of interfaces will be found in [2] or [4] where various implementations of the readers and writers paradigm are explained.

The operations of a synchronized interface are denoted by subprograms. Thus we might have

```
package Pkg is
  type Server is synchronized interface;
  procedure Q(S: in out Server; X: in Item) is abstract;
end Pkg;
```

We can then implement the interface by a task type or by a protected type. This introduces several different ways of implementing the operation Q. It can be by an entry, or by a protected procedure or by a normal procedure. For example using a task type we might have

```
package TP1 is
  task type TT1 is new Server with
                  -- Q implemented by entry
    entry Q(X: in Item);
  end TT1;
end TP1;
```

or

```
package TP2 is
  task type TT2 is new Server with
                          -- Q implemented by a normal procedure
  end TT2;
  procedure Q(S: in out TT2; X: in Item);
end TP2;
```

Similarly using a protected type we might have

```
package PP1 is
  protected type PT1 is new Server with
                  -- Q implemented by entry
    entry Q(X: in Item);
    ...
  end PT1;
end PP1;
```

or

```
package PP2 is
  protected type PT2 is new Server with
                          -- Q implemented by a protected procedure
    procedure Q(X: in Item);
    ...
  end PT2;
end PP2;
```

or

```
package PP3 is
  protected type PT3 is new Server with
                           -- Q implemented by a normal procedure
    ...
  end PT3;
  procedure Q(X: In out PT3; X: in Item);
end PP3;
```

So the interface Server could be implemented in many different ways. And as usual we could dispatch to any of the implementations. We could have

```
Server_Ptr: access Server'Class := ...
...
Server_Ptr.Q(X => An_Item);
```

and this will dispatch to the implementation of Q concerned.

So a call of Q could end up as a call of an entry in a task, an entry in a protected object, a protected procedure in a protected object, or an ordinary procedure.

Two curious situations arise. One concerns timed calls. We could write a timed call such as

```
select
  Server_Ptr.Q(An_Item);
or
  delay Seconds(10);
end select;
```

and this will always be acceptable. It will dispatch to the appropriate operation. If it is an entry then it will be a timed call. But if it is not an entry then no time-out is possible and so by default the call will always go ahead.

The other curious situation concerns requeue. In this case there is no obvious default action. It is not possible to requeue a procedure call since there is no queue on which to hang it.

The first proposal to do something about this was simply not to allow requeue at all on interfaces. And indeed this was the solution adopted in Ada 2005.

However, this is not really acceptable as explained in [3]. The next idea was to raise some exception if it turned out that the destination was not an entry. But this was considered unsatisfactory.

So it was concluded that if we do a requeue then it must be statically checked that it will dispatch to an entry so that the requeue is possible. The next proposal was that there should be a pragma Implemented giving requirements on the operation. Thus we might have

```
procedure Q(S: in out Server; X: in Item) is abstract;
pragma Implemented(Q, By_Entry);
```

and the compiler would ensure that all implementations of the interface Server did indeed implement Q by an entry so that requeue would always work. The other possible values for the pragma were By_Protected_Procedure and By_Any.

The world changed when the notion of an aspect was invented and so after much discussion the final solution is that we there is now an aspect Synchronization so we write

```
procedure Q(S: in out Server; X: in Item) is abstract
  with Synchronization => By_Entry;
```

and we are now assured that we are permitted to do a requeue on Q for any implementation of Server. The other possible values for the aspect Synchronization are By_Protected_Procedure and Optional.

In summary, if the property is By_Entry then the procedure must be implemented by an entry, if the property is By_Protected_Procedure then the procedure must be implemented by a protected procedure, and if the property is Optional then it can be implemented by an entry, procedure or protected procedure. Naturally enough, the aspect cannot be given for a function.

There are a number of rules regarding consistency. The aspect Synchronization can be applied to a task interface or protected interface as well as to a synchronized interface. However, if it is applied to a task interface then the aspect cannot be specified as By_Protected_Procedure for obvious reasons. If a type or interface is created by inheritance from other interfaces then any Synchronization properties are also inherited and must be consistent. Thus if one is By_Entry then the others must also be By_Entry or Optional.

A final minor improvement mentioned in the Introduction concerns renaming. Since the days of Ada 83 it has been possible to rename an entry as a procedure thus

```
procedure Write(X: in Item) renames Buffer.Put;
```

where Put is an entry in a task Buffer. But in Ada 83 it was not possible to do a timed call using Write. This was corrected in Ada 2005 which allows a timed call on a renaming. Similarly, when requeue was introduced in Ada 95, it was not possible to do a requeue using Write. This anomaly is corrected in Ada 2012. So now both timed calls and requeue are permitted using a renaming of an entry.

References

[1] ISO/IEC JTC1/SC22/WG9 N498 (2009) Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of Amendment 2 to ISO/IEC 8652.

[2] John Barnes (2006) *Programming in Ada 2005*, Addison-Wesley.

[3] A. Burns and A. Wellings (2007) *Integrating OOP and Tasking – the missing requeue*, from IRTAW 2007, www.ada-auth.org/ai-files/grab_bag/requeue.pdf.

[4] John Barnes (2008) *Ada 2005 Rationale*, LNCS 5020, Springer-Verlag.