Interrupts, Timing Events and Dispatching Domains

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

Abstract

This paper proposes extensions to the Ada interrupt handling facilities to allow the dispatching domain of handlers to be set, where this is supportable by the underlying platform.

1 Introduction

It is useful in real-time systems to distinguish between two forms of computation that occur at run-time: tasks and event handlers. A real-time task (or thread) is a long-lived entity with state and periods of activity and inactivity. While active, it competes with other tasks for the available resources – the rules of this competition are captured in a scheduling or dispatching policy (for example, fixed priority or EDF). A real-time task can be released by the passage of time or by an event signalled by another task or the environment in which the program executes.

Events handlers can similarly be released by the passage of time or be event triggered. However, in comparison with real-time tasks, an event handler is usually a short-lived, stateless, one-shot or periodic computation. Its execution is, at least conceptually, immediate; and having completed it has no lasting direct affect other than by means of changes it has made to the permanent state of the system. For example, if a central heating system must come on at 7.00am then the control system needs a clock and a way of postponing execution until that clock says 7.00am. Using tasks, the only way to deliver this coordination is to have the task delay until 7.00am and then turn the heating on. In this and other situations, this concurrency overhead is unnecessary and inefficient. Hence events and event handlers have an important role to play.

When an event occurs, it is said to be *triggered*; other terms used are *fired*, *invoked* or *delivered*. The event handling code normally does not contain any synchronisation calls that could lead to it becoming suspended; the handler runs to completion. Where these requirements cannot be guaranteed, it is necessary to schedule the event handlers rather than execute them immediately.

Prior to Ada 95, Ada only supported tasks (and interrupts were viewed as hardware-generated entry calls). However since then, the language has been providing an increasing level of support for events and their handlers. Ada 95 introduced protected types and the idea of a protected action that could, potentially, be executed by the environment in response to interrupts. These interrupt handlers can be viewed as event handlers, where the corresponding events are the associated interrupts. Ada 2005 extended the model with the introduction of timing events and execution-time clocks (and timers) and group budget timers. Common to all these is the notion of a handler that is called asynchronously by the Ada run-time systems usually at a high software or interrupt-level priority.

Ada 2012 has provided improved support for multiprocessor applications. In particular, it introduces the notion of a dispatching domain, which can be used to constrain the set of processors on which tasks are allowed to execute. The Ada 2012 mechanisms allow fully partitioned, cluster-based and globally scheduled applications to be configured. Hence the programmer has control over how the workload of an application can be distributed across the execution platform. Unfortunately, in designing the multiprocessor support we focused on the tasking model and paid little attention to where the event handlers execute. For many embedded system, the load imposed from, say, handling interrupt can be significant. Indeed, it is for this reason that many multiprocessor systems allow interrupts to be targeted at particular processors. For example, the ARM Corex A9-MPCore supports the ARM Generic Interrupt Controller[1]. This allows a target list of CPUs to be specified for each hardware interrupt. Software generated interrupts can also be sent to the list or set up to be delivered to all but the requesting CPU or only the requesting CPU. Currently, there are no mechanisms provided by Ada that allow the execution of Ada-level interrupt handlers (protected procedures) and any other event handlers to be constrained to a set of processors. `Ada.Interrupts` does, however, allow the processor on which an interrupt is serviced to be determined.

This short paper proposes some simple extensions to package `Ada.Interrupts` and `Ada.Interrupts.Names` that allow the dispatching domains (and potentially a single processor within a domain) for interrupts to be specified. This will allow, at least, a coarse level of control over where event handlers are executed.

In Section 2, the Ada support for multiprocessors is briefly reviewed, along with those parts of the language that are related to event handling. Then in Section 3, we present our proposed additions to the language. Finally, we present our conclusions.

2 Multiple Processors and Event Handling in Ada 2012

2.1 Multiprocessors

Although multiprocessors are becoming prevalent, there are no agreed standards on how best to address real-time demands. This is partly because multiprocessor and multicore architectures are

[1] See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegbfjhf.html

evolving rapidly, and operating system standards (in particular the suite of POSIX Standards) have yet to catch up. Most real-time operating systems nowadays support multiprocessor platforms in some guise. For example, the RTEMS operating system does not dynamically move threads between CPUs. Instead it provides mechanisms whereby they can be statically allocated at link time. In contrast, QNX's Neutrino distinguishes between "hard thread affinity" and "soft thread affinity". The former provides a mechanism whereby the programmer can require that a thread be constrained to execute only on a set of processors (indicated by a bit mask). With the latter, the kernel dispatches the thread to the same processor on which it last executed (in order to cut down on preemption costs). Other operating systems provide similar facilities. For example, IBM's AIX allows a kernel thread to be bound to a particular processor [2]. In addition, AIX enables the set of processors (and the amount of memory) allocated to the partition executing an application to be changed dynamically.

The Ada 2012 Reference Manual allows a program's implementation to be on a multiprocessor system and provides direct support that allows programmers to partition their tasks onto the processors in the given system. The packages shown in Figures1 and 2 allows the group of CPUs to be partitioned into a finite set of non-overlapping **dispatching domains**. A dispatching domain represents a series of processors on which a task may execute. Each processor is contained within exactly one dispatching domain. One dispatching domain is defined to be the *System* dispatching domain; the Ada environment task and any created from that task are allocated (by default) to the *System* dispatching domain.

Subprograms are defined to allow new dispatching domains to be created. Tasks can be assigned to a dispatching domain and be scheduled globally within that dispatching domain; alternatively they can be assigned to a dispatching domain and restricted to executing on a specific CPU within that dispatching domain. Tasks cannot be assigned to more than one dispatching domain, or restricted to more than one CPU. Note that AI 0033 modified the Ada 2012 package to add CPU sets.

Figure 1 shows the parent unit for all multiprocessor facilities, and gives a range for an integer representation of each CPU. Note the value 0 is used to indicate `Not_A_Specific_CPU`; so the CPUs are actually numbered from one. The function `Number_Of_CPUs` will, for any execution of the program, return the same value. In effect the number of CPUs is a constant and reflects the number of available processors at system start up.

As well as being able to call the subprograms in Figure 2, a task can have its CPU and dispatching domain set using a language-defined representation aspects at the task's declaration time.

2.2 Event Handling

In Ada 2012, the following event handlers are called asynchronously by the Ada run-time or the execution-time environment:

[2] see http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.basetechref/doc/basetrf1/-bindprocessor.htm

```
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;
```

Figure 1. The Ada Multiprocessors package

```
with Ada.Real_Time;
with Ada.Task_Identification;
package System.Multiprocessors.Dispatching_Domains is
  Dispatching_Domain_Error : exception;
  type Dispatching_Domain (<>) is limited private;
  type CPU_Set is array(CPU range <>) of Boolean;
  System_Dispatching_Domain : constant Dispatching_Domain;

  function Create (Set : CPU_Set) return Dispatching_Domain;
  function Create (First, Last : CPU) return Dispatching_Domain;
  function Get_First_CPU (Domain : Dispatching_Domain) return CPU;
  function Get_Last_CPU (Domain : Dispatching_Domain) return CPU;

  function Get_CPU_Set  (Domain : Dispatching_Domain) return CPU_Set;
  function Get_Dispatching_Domain
      (T : Ada.Task_Identification.Task_Id :=
       Ada.Task_Identification.Current_Task)
         return Dispatching_Domain;
  procedure Assign_Task(Domain : in out Dispatching_Domain;
                    CPU : in CPU_Range := Not_A_Specific_CPU;
                    T : in Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);
  procedure Set_CPU(CPU : in CPU_Range;
                T : in Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task);
  function Get_CPU(T : Ada.Task_Identification.Task_Id :=
              Ada.Task_Identification.Current_Task)
                 return CPU_Range;
  procedure Delay_Until_And_Set_CPU
      (Delay_Until_Time : in Ada.Real_Time.Time;
       CPU : in CPU_Range);
private
... -- not specified by the language
end System.Multiprocessors.Dispatching_Domains;
```

Figure 2. The Ada Dispatching_Domains package

- interrupt handler – represented by

  ```
  type Parameterless_Handler is access protected procedure
  ```

- timing-event handler – represented by

  ```
  type Timing_Event_Handler is access protected procedure
                          (Event :  in out Timing_Event)
  ```

- execution-timer handler – represented by

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

- group-budget handler – represented by

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

As can be seen, all handlers are defined to be protected procedures. The incremental development of the model means that, unfortunately, there is no common root type and no commonality in the names of the operations that can be called to get and set the handlers. For example: for interrupt handlers it is :

```
package Ada.Interrupts is

  function Is_Attached(Interrupt : Interrupt_Id) return Boolean;

  function Current_Handler(Interrupt : Interrupt_Id)
                           return Parameterless_Handler;

  procedure Attach_Handler(New_Handler : Parameterless_Handler;
                           Interrupt : Interrupt_Id);

  procedure Exchange_Handler(
            Old_Handler : out Parameterless_Handler;
            New_Handler : Parameterless_Handler;
            Interrupt : Interrupt_Id);

  procedure Detach_Handler(Interrupt : Interrupt_Id);

  function Reference(Interrupt : Interrupt_Id)
           return System.Address;

  function Get_CPU (Interrupt : Interrupt_Id)
            return System.Multiprocessors.CPU_Range;
end Ada.Interrupts;
```

and for timing events it is:

```
package Ada.Real_Time.Timing_Events is
  ..
  procedure Set_Handler(Event : in out Timing_Event;
        At_Time : Time; Handler: Timing_Event_Handler);
  procedure Set_Handler(Event : in out Timing_Event;
```

```
        In_Time: Time_Span; Handler: Timing_Event_Handler);
   function Is_Handler_Set(Event : Timing_Event)
        return Boolean;
   function Current_Handler(Event : Timing_Event)
          return Timing_Event_Handler;
   procedure Cancel_Handler(Event : in out Timing_Event;
            Cancelled : out Boolean);

end Ada.Real_Time.Timing_Events;
```

Any call to a handler ultimately comes via an interrupt. While `Ada.Interrupts.Get_CPU` allows the CPU on which the interrupt is handled to be returned[3], it not possible to set the dispatching domain for the handler associated with the interrupt. Where the handler of an interrupt can be configured by a run-time call, it ought to be possible for the Ada program to make that call.

3 Proposals

The proposal to be discussed at IRTAW 2015 is that

- `Ada.Interrupts` should provide mechanisms to set the dispatching domain and potentially an individual processor for an interrupt. The standard exception (`Dispatching_Domain_Error`) should be raised if the operation is not supported on a particular platform.
- `Ada.Interrupts.Names` should declare standard names for all the reserved interrupts required by the Ada run-time, e.g. the clock interrupts that service timing events and those that allow tasks to be released when a delay (or delay until) time has expired.
- There should be documentation required that indicates which reserved interrupts result in which of the event handlers being executed.
- Where extra tasks are introduced to execute the event handlers, the tasks should have the same locality as the associated interrupt.

4 Conclusions

In this short paper we have identified a gap in the Ada facilities for supporting multiprocessors. This gap is that Ada does not allow the dispatching domains of its event handlers to be specified. The call to all event handlers in Ada originate from interrupt handlers. We have proposed a simple extension that allows the dispatching domain of an interrupt handler to be set. If this is not supportable by the underlying platform a standard exception is raised at run-time.

[3]Note that the assumption here is that the interrupt handler runs on a single CPU not globally within a dispatching domain. Although, of course a returned value of 0 means *not a specific CPU*.