SPARK

Quick Reference 3

RavenSPARK Patterns

RavenSPARK and the Examiner

To use the Ravenscar profile with the Examiner, provide the profile option with a value of *ravenscar* (which can be abbreviated to *r)*. The default value of profile is *sequential* (which can be abbreviated to s).

Typical usages:

```
spark -pr=ravenscar p.adb
spark –pr=r @allunits
```

Periodic Tasks

A periodic task is a task that runs at set intervals. The intervals are controlled by a *delay until* statement that *must* have an absolute (not relative) time as its argument. The initial time can be obtained from the Ada.Real_Time.Clock or, more usually, from some program-wide start time provided by an "Epoch" package (see below).

```
task type T <any discriminants go here>
--# global ...;
--# derives ...; -- describe the effect of repeated
execution of the task body
is
   pragma Priority (...); -- or Interrupt_Priority
end T;

task body T
is
   Release_Time: Ada.Real_Time.Time :=
                            Epoch.T_Start;

   Period : Ada.Real_Time.Time_Span :=
                            Epoch.T_Period;
begin
   <initialisation code>
   loop
      delay until Release_Time; -- deterministic release
      -- perform the periodic action required
      Do_Periodic_Work;
      -- calculate next time to run
      Release_Time := Release_Time + Period;
   end loop;
end T;
```

Protected Objects

In RavenSPARK we allow package own variables to be marked as being *protected*. In the example below, calls to the entry Wait will be blocked by *TheBarrier*. A protected sequence of statements may not call subprograms that can suspend or delay execution. Where an object *may* suspend a task, it must be annotated to indicate this with the *suspendable* property. A protected type may have only one entry.

```
package Q
--# own protected PO : PT ( priority => 10,
                            suspendable);
is
   type Data is ...;

   protected type PT is
      pragma Priority (10);

      procedure Signal (D: in Data);
      --# global in out PT;
      --# derives PT from PT, D; -- "PT" here refers
                       -- to "this instance of type"

      entry Wait       (D: out Data);
      --# global in PT;
      --# derives D from PT;

   private
      TheData    : Data    := ...;
      TheBarrier : Boolean := False;
   end PT;

   PO : PT;

end Q;
```

The body for the type PT might look something like this. Refined global and derives annotations must be given using the protected elements.

```
protected body PT is
   entry Wait (D : out Data) when TheBarrier
   --# global in TheData; out TheBarrier;
   --# derives D          from TheData &
   --#         TheBarrier from ;
   is
   begin
     D := TheData;
     TheBarrier := False;
   end GetDataWhenReady;

   procedure Signal (D : in Data)
   --# global out TheBarrier;
   --# derives TheData    from D &
   --#         TheBarrier from ;
   is
   begin
     TheData := D;
     TheBarrier := True;
   end Release;
end PT;
```

Sporadic Tasks

A sporadic task is a task that is released by some external stimulus rather than by the passing of time.

Release by Suspension Object

A suspension object is of the predefined type *Suspension_Object* in the *Ada.Synchronous_Task_Control* package which can be set to True or False. The suspension object would be an own variable with the annotation:

```
--# own protected DataReady (Suspendable);
```

The procedure *Suspend_Until_True* will suspend the task until the *Set_True* procedure is called by another task.

```
DataReady :
Ada.Synchronous_Task_Control.Suspension_Object;

task body ProcessWhenReady
is
begin
   loop
      -- wait until there is something to do
      Ada.Synchronous_Task_Control.
                        Suspend_Until_True
(DataReady);
      -- do it
      P.ProcessTheState;
   end loop;
end ProcessWhenReady;
```

Release by Entry

Using the protected package *Q* defined earlier. The task will be suspended on the call to Wait, until *Signal* is called. Note that only one task may suspend on any one entry. The task type definition should indicate that the task may suspend, and on which object, with a declares statement:

```
--# declare suspends => Q.PO;
```

A suspending task body:

```
task body T is
   My_Data : P.Data;
begin
   loop
      Q.PO.Wait (My_Data); -- suspend until data available
      Operate_On (My_Data);
   end loop;
end T;
```

Interrupt Handlers

An interrupt handler is parameterless protected procedure which is executed not by a procedure call statement but by an external event signalled by an interrupt. In the example below, the priority must be in the range System.Interrupt_Priority

```
--# inherit SomePackage;
package Interrupts
--# own protected Handler : PT
--#    (priority => 31,
--#     interrupt => (Event => UserSuppliedName));
is
private
   protected type PT is
      pragma Interrupt_Priority (31);

      procedure Event;
      --# global in out SomePackage.State;
      --# derives SomePackage.State from
      --#                        SomePackage.State;
      pragma Attach_Handler (Event, 42);-- make it a
                                        --handler
   end PT; -- no protected elements declared
end Interrupts;

package body Interrupts
is
   Handler : PT;
   protected body PT is separate;
end Interrupts;

with SomePackage;
separate (Interrupts)
protected body PT is
   procedure Event
   is
   begin
      SomePackage.DoWork;
   end Event;
end PT;
```

Thread Safe Polled Input Port

This example uses protected elements to provide thread safe access to the raw input port (RawPort). The *protects* statement indicates which variable is being protected.

```
package SharedPort
--# own           in RawPort;
--#     protected in SafePort : PortType
--#        (priority => 10, protects => RawPort);
is
   function Read return Natural;
   --# global SafePort;

private
   protected type PortType is
      pragma Priority (10);

      function PRead return Natural;
      --# global PortType;
   end PortType;
end SharedPort;
```



```
package body SharedPort
is
   RawPort : Natural;
   for RawPort'Address use 16#FFFF_FFFF#;
   pragma Volatile (RawPort);

   SafePort : PortType;

   protected body PortType is
      function PRead return Natural
      --# global RawPort;
      is
         ReadLocal : Natural;
      begin
         ReadLocal := RawPort;
         if not ReadLocal'Valid then
            ReadLocal := 0;
         end if;
         return ReadLocal;
      end PRead;
   end PortType;

   function Read return Natural
   is
   begin
      return SafePort.PRead;
   end Read;
end SharedPort;
```

Interrupt Driven Input Port

An interrupt can be used in place of polling processes to drive safe access to input ports.

```
package InterruptPort
--# own           in RawPort;
--#     protected    SafePort : PortType
--#        (priority => 31, protects => RawPort,
--#         interrupt, suspendable);
is
   procedure Read (X : out Data);
   --# global in out SafePort;
   --# derives X, SafePort from SafePort;
   --# declare suspends => SafePort;
private
   protected type PortType is
      pragma Interrupt_Priority (31);

      procedure DataReady;
      --# global in out PortType;
      --# derives PortType from PortType;
      pragma Attach_Handler (DataReady, 5);

      entry PRead (X : out Natural);
      --# global in out PortType;
      --# derives X, PortType from PortType;
   private
      Ready : Boolean := False;
      TheData     : Natural := 0;
   end PortType;
end InterruptPort;
```

In the body below, the interrupt removes the barrier for the entry, allowing the input port to be read.

```
package body InterruptPort
is
   RawPort : Natural;
   for RawPort'Address use 16#FFFF_FFFF#;
   pragma Volatile (RawPort);
   SafePort : PortType;

   protected body PortType is
      procedure DataReady
      --# global out Ready, TheData; in RawPort;
      --# derives Ready from &
      --#         TheData     from RawPort;
      is
         ReadLocal : Natural;
      begin
         TheData := RawPort;
         Ready := True;
      end DataReady;

      entry PRead (X : out Natural) when Ready
      --# global out Ready; in TheData;
      --# derives Ready from &
      --#         X     from TheData;
      is
      begin
         X := TheData;
         Ready := False;
      end PRead;
   end PortType;

   procedure Read (X : out Natural)
   is
   begin
      SafePort.PRead (X);
   end Read;
end InterruptPort;
```

In both this, and the previous (polled input) example, the raw input port is considered a *virtual* protected variable. The protected object is the only object that may access it, resulting in it behaving exactly as if it were a protected element of the type. The protects property indicates this relationship.

Epoch Package

An “Epoch” package can be used to define a reference time from which to co-ordinate task startup:

```
with Ada.Real_Time;
use type Ada.Real_Time.Time;
--# inherit Ada.Real_Time;
package Epoch is
   StartTime : constant Ada.Real_Time.Time :=
                              Ada.Real_Time.Clock;
   T_Start : constant Ada.Real_Time.Time :=
      StartTime + Ada.RealTime.Milliseconds (10);
   T_Period : constant Ada.Real_Time.Time_Span :=
      Ada.Real_Time.Milliseconds (50);
end Epoch;
```