Tool Development and Support

RavenSPARK Design for the Mine Pump Case Study

S.P0468.76.21
Issue: 1.5
Status: Definitive
22 November 2005

Originator

SPARK Team

Approver

SPARK Team Line Manager

Contents

1 Introduction 3

2 The Mine Pump 4
2.1 Mine Pump Control Panel 4

3 Architecture 5

4 Design 6
4.1 Inputs 6
4.2 Outputs 9
4.3 Partition Wide Flow Relations 10
4.4 Start-up 12
4.5 Normal Operation 13
4.6 Scheduling 15

A Mine Pump Requirements 16

B Code 19

Document Control and References 43
Changes history 43
Changes forecast 43
Document references 43

1 Introduction

The mine pump example is taken from [1] and is used here as an example of a RavenSPARK program. The problem is a typical real-time application with actuators being set as a result of various sensors in accordance with some requirement.

A sequential SPARK solution to this problem would have required the use of a single cyclic scheduler. The application monitors various inputs with various response times which would render this approach non-trivial. The introduction of the Ravenscar profile to the SPARK language now enables us to decompose the problem into a number of threads resulting in less coupled, more cohesive, understandable, extendable and maintainable solution.

The program aims to illustrate many of the RavenSPARK features and how they interrelate to form a complete system. The solution makes use of: passive sensors (ports), active sensors (interrupt handlers), periodic and aperiodic tasks, protected and suspension objects. We use techniques such as protected refinement, interrupt streams and visit synchronisation issues.

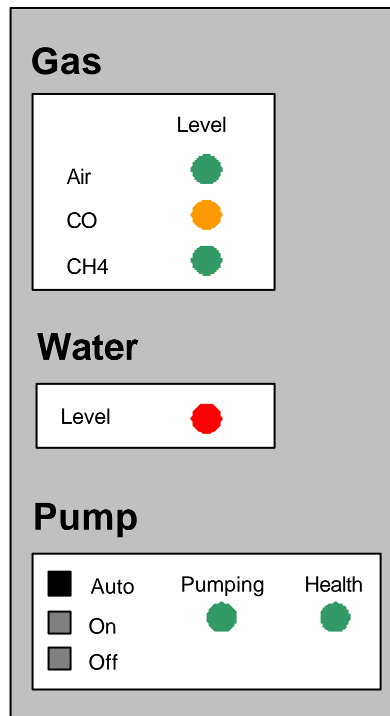
Our interpretation of the requirements for the mine pump is given in Appendix A. The design method and notation used follows that described in [2] with the extension that variable and boundary packages with bold outlines provide shareable, protected state.

2 The Mine Pump

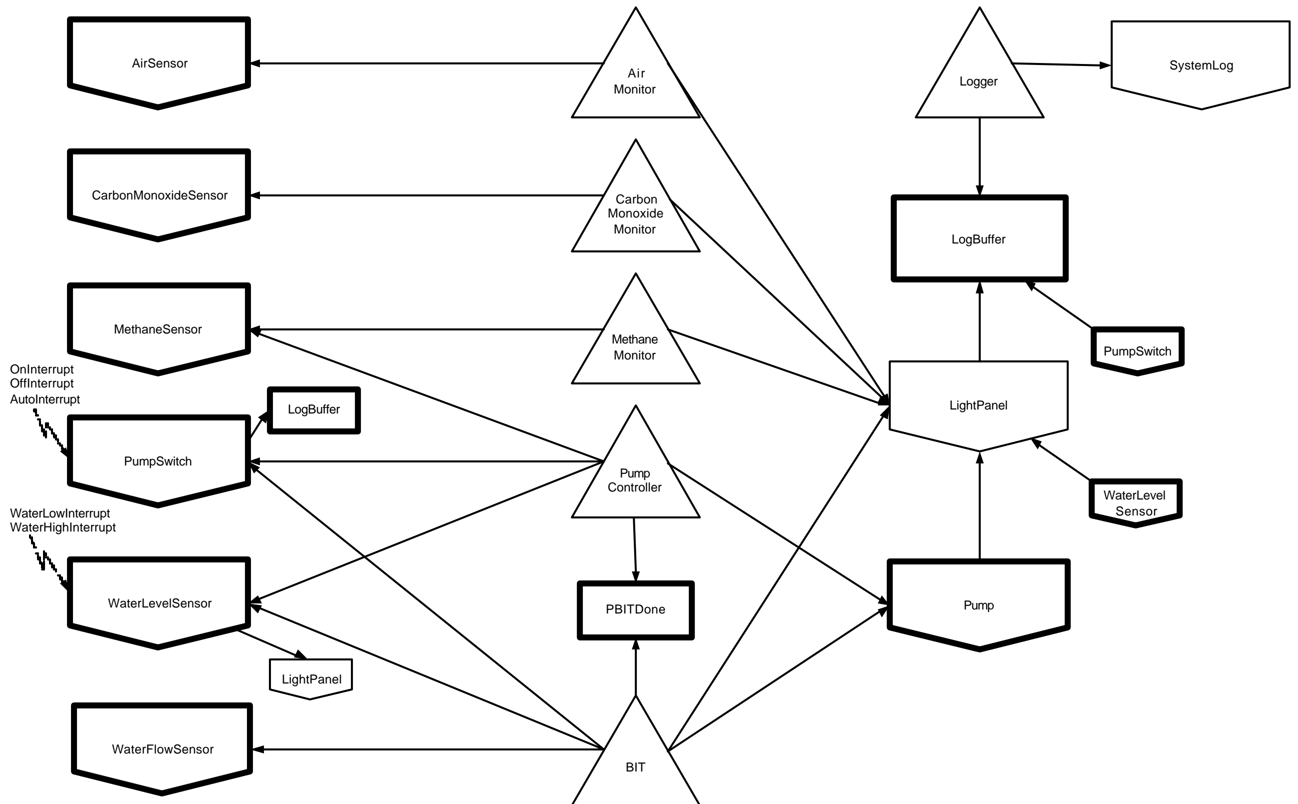
The basic functional requirement of the mine pump is that it pumps water out of a mine when the water level gets too high. For safety reasons the pump must not operate when the methane level is too high and the operator must be able to manually override the pump. There must be a visual representation of system state and all critical events must be recorded in an external log.

2.1 Mine Pump Control Panel

The mine pump control panel is made up of a pump switch consisting of three radio buttons which manually set the pump state and a set of lights providing feedback on the pump and its environment. A green light signifies a normal situation, a red light an abnormal situation and an amber light an unknown situation (for example a broken sensor).



3 Architecture



4 Design

The system architecture is given in section 3.

4.1 Inputs

The system has seven inputs: 3 gas sensors, 2 water sensors, the pump switch and the pump state. Each sensor is either active or passive.

4.1.1 Passive Sensors

The passive sensors (AirSensor, CarbonMonoxideSensor, WaterFlowSensor and MethaneSensor) are implemented using thread-safe I/O ports as described in [3]. Take for example the AirSensor:

```
package AirSensor
--# own           in RawSensor;
--#     protected in State :
--#        SensorPT (Priority => System.Interrupt_Priority'Last,
--#                  Protects => RawSensor);
```

The `AirSensor` package has two items of state that, together, illustrate an example of protected refinement. Protected refinement is the ability to render unprotected state protected by ensuring that all access to it are via a protected object. It is achieved via the protects property and gives the protected object sole access to all the items it protects (in this case only one). A semantic error would be raised if `RawSensor` (known as a *virtual protected* element) appeared in a global list outside the protected object. Note that the normal SPARK rules for own variable refinement apply for protected own variable refinement in that if the protected own variable has a mode then all its elements must be virtual protected elements with the same mode. So in this case we can declare the protected own variable `State` to be mode in.

Generally protected refinement is used to allow a memory-mapped variable to be accessible by more than one thread. However, in this case, the air monitor is the only thread that accesses the air sensor. The reason for the protected object is that the operation returns a timestamp for the reading. In order to guarantee that the reading of the raw sensor and the time is atomic we have placed them inside a protected object with maximum priority. Note that the package used to get a system timestamp is not Ada.Real_Time which is used for scheduling purposes. The Timestamp package exports an own variable ClockTime which represents the current time. In reality the system would have to provide a mechanism for the user to enter the current time. How time is tracked would depend on the level of support from the compiler vendor.

Although RavenSPARK forces the declaration of the protected type to be in the package specification we can still place it in the private part and make it accessible via exported operations. The object itself can be located in the package body.

The other passive input to the system is the pump state. This is modelled as a normal in stream as it is accessed by one thread only.

4.1.2 Active Sensors

The active sensors (WaterLevelSensor and PumpSwitch) are implemented using interrupt handlers as described in [3]. Take for example the PumpSwitch.

```
package PumpSwitch
--# own protected State :
--#        PumpSwitchPT
--#           (Priority => System.Interrupt_Priority'Last,
--#            Interrupt)
```

The package PumpSwitch declares one item of protected state, `State`. The interrupt property is used to identify that the protected object is responsible for handling interrupts. When an interrupt occurs the protected object's internal state is updated to reflect the state of the switch. The package can then be polled using an exported function to get the contents of the protected object's state.

Note here that we have used the minimal form of the interrupt property. This causes the interrupt to be represented by the state variable PumpSwitch.`State`. At the partition level this variable will appear as an import for anything affected by the interrupt. It will also appear to be derived from itself.

Note that the priority of a protected object that declares interrupt handlers must be in the range `System.Interrupt_Priority`. Again, the protected type is hidden in the private part of the package specification and made its operation made available via an exported function.

```
protected type PumpSwitchPT
is

   pragma Interrupt_Priority (System.Interrupt_Priority'Last);

   procedure AutoInterruptHandler;
   --# global in     AdaCalendar.ClockTime;
   --#        in out LogBuffer.State;
   --#           out PumpSwitchPT;
   --# derives PumpSwitchPT    from  &
   --#         LogBuffer.State from *,
   --#                             AdaCalendar.ClockTime;
   pragma Attach_Handler (AutoInterruptHandler, 3);

   -- similar for other interrupts

   function ProtectedGetState return Value;
   --# global in PumpSwitchPT;

private
   CurrentState : Value := Undefined;
end PumpSwitchPT;
```

Note that the internal state is initialised to undefined. An interrupt must occur for the state to reflect the real pump switch state. We address this problem in the section on system start-up.

The interrupt property has an extended form that allows you to map names onto each interrupt by means of a named association using the handler names as the formals. We refer to the actuals as *interrupt stream variables* because they represent a notional stream of interrupts being picked up by the handler. We have used this form for the water level sensor to illustrate the point.

```
package WaterLevelSensor
--# own protected State :
--#        SensorPT
--#         (Priority => System.Interrupt_Priority'Last,
--#          Interrupt => (WaterIsHighInterruptHandler =>
--#                              WaterLevelInterrupt,
--#                           WaterIsLowInterruptHandler  =>
--#                              WaterLevelInterrupt));
```

The interrupt stream variable is implicitly declared by this mapping and will appear only at the partition wide level. The advantage of this mechanism is that you can control the granularity of the interrupts at the partition wide annotation level. In our case we have specified the same interrupt stream variable for each handler and so we end up with these annotations at the partition level:

```
--# LightPanel.WaterLevelLight from
--#        WaterLevelSensor.WaterLevelInterrupt &
--#
--# Pump.Actuator                from
--#        WaterLevelSensor.State,
--#        WaterLevelSensor.WaterLevelInterrupt,
--#        ... &
--# WaterLevelSensor.State      from,
--#        WaterLevelSensor.WaterLevelInterrupt
```

The variable `WaterLevelSensor.State` appears as an import for the pump actuator because the `PumpController` task reads the state to determine the actuator state. The water level light however is set directly from the interrupt handler and so the state plays no part in it. Also note that `WaterLevelSensor.State` is now derived from the interrupt stream variable which makes slightly more sense than being derived from itself. The point here is that we wanted to show that the pump was dependent on the water level sensor and not its individual interrupts. Had we had an alarm bell that sounded when the level got too high but not when too low we could have specified distinct stream variables `WaterLowInterrupt` and `WaterHighInterrupt` and our partition wide annotation would have been:

```
--# LightPanel.WaterLevelLight from
--#        WaterLevelSensor.WaterLowInterrupt,
--#        WaterLevelSensor.WaterHighInterrupt &
--#
--# Pump.Actuator                from
--#        WaterLevelSensor.State,
--#        WaterLevelSensor.WaterLowInterrupt,
--#        WaterLevelSensor.WaterHighInterrupt
--#        ... &
--#
--# Bell.Actuator from WaterLevelSensor.WaterHighInterrupt
```

We have thus proved that the requirement on the bell has been satisfied.

4.2 Outputs

The outputs are the pump actuator, the light panel and the system log. The LogBuffer serves as an intermediate output.

4.2.1 Pump

The pump actuator is implemented as a stream variable.

```
package Pump
--# own protected out Actuator :
--#        ActuatorPT (Priority => System.Interrupt_Priority'Last);
```

The pump actuator is only accessed by one task and the protection is used, again, to synchronise the time with the actuator command in the system log.

4.2.2 LightPanel

The light panel is used to convey information to the operator. Each light is represented by an external variable. This ensures that the partition wide flow annotation shows the relationship between each light and its sensors. We can hence ensure that each light is derived from the correct sensors. The package is accessed by many tasks but each item of state is accessed by only one task so no protection is required.

4.2.3 SystemLog

The SystemLog records events and their associated times. The package exports a call that writes to the log. Writing to the log is potentially blocking and must be identified by the `declare delay` annotation.

```
package SystemLog
--# own out State;
is

   procedure Write (Data       : in LogBuffer.BufferEntry;
                    DataMissed : in Boolean);
   --# global out State;
   --# derives State from Data,
   --#                   DataMissed;
   --# declare delay;

end SystemLog;
```

The fact that the call is potentially blocking means that we cannot call it directly from a protected object. To circumvent this we have an intermediate output `LogBuffer` that logs all events to a cyclic buffer.

4.2.4 LogBuffer

The LogBuffer stores the events in a cyclic buffer. These events are read by the logger task (at low priority) and written to the SystemLog.

```
protected type Log
is
   pragma Interrupt_Priority (System.Interrupt_Priority'Last);

   procedure ProtectedWrite (Data : in BufferEntry);
   --# global in out Log;
   --# derives Log from *,
   --#                  Data;

   entry ProtectedRead (Data       : out BufferEntry;
                        DataMissed : out Boolean);
   --# global in out Log;
   --# derives Data,
   --#         DataMissed,
   --#         Log        from Log;

private
   TheBuffer  : Buffer  := EmptyBuffer;
   ReadIndex  : Index   := Index'First;
   WriteIndex : Index   := Index'First;
   HasEntries : Boolean := False;
   DataOverwittenSinceLastRead : Boolean := False;
end Log;
```

The LogBuffer can be written at any time but read only when data is available. The read operation is therefore implemented as an entry. The accessor operation must therefore declare a suspends property.

```
  procedure Read (Data       : out BufferEntry;
                  DataMissed : out Boolean);
   --# global in out State;
   --# derives State,
   --#         Data,
   --#         DataMissed from State;
   --# declare suspends => State;
```

The DataMissed parameter indicates that data has been lost due to overflow since the last read.

4.3 Partition Wide Flow Relations

Having defined the system inputs and outputs we can now generate the partition wide flow relation. This is a very important part of the design process and should be done once all the inputs and outputs have been defined.

We know the pump actuator is dependent on the pump switch, methane and water levels. At the partition level, we're not interested in the various interrupts that make up this state and would not want to see them cluttering up the flow relation. The `BIT.PBITDone` suspension object appears in this annotation as the pump actuator is never set until it is set true. This is discussed in more detail in section 4.4.

```
--# derives Pump.Actuator from
--#            MethaneSensor.State,
--#            WaterLevelSensor.State,
--#            PumpSwitch.State,
--#            WaterLevelSensor.WaterLevelInterrupt,
--#            BIT.PBITDone &
```

The way we modelled the state in the lighting package is worth a closer look. One solution would have been to use protected refinement and declare a single piece of protected state and have it protect the individual light states. Had we done this, the flow relation for the light panel would have been `derives LightPanel.State from` <all the sensors>. This may have been acceptable but we would like to show (for safety reasons) that each light is derived from the correct sensor. To do this we don't abstract the light state and hence each individual state appears at the partition level. Another consequence is that the state does not need to be protected as each light is set by only one thread and hence we have reduced the number of critical regions in the program.

```
--#         LightPanel.AirLight from
--#            AirSensor.State &

--#         LightPanel.CarbonMonoxideLight from
               CarbonMonoxideSensor.State &

--#         LightPanel.MethaneLight from
--#            MethaneSensor.State &

--#         LightPanel.WaterLevelLight from
               WaterLevelSensor.WaterLevelInterrupt &

--#         LightPanel.PumpPumpingLight from
--#            WaterFlowSensor.State &
```

The pump health light is set by comparing the water flow state to the pump state. E.g. if the pump thinks it's on and no water is flowing it must be broken.

```
--#         LightPanel.PumpHealthLight from
--#            WaterFlowSensor.State,
--#            Pump.State &
```

The system log is generated from all the inputs and the clock for the timestamps.

```
--#         SystemLog.State from
--#            LogBuffer.State,
--#            AirSensor.State,
--#            CarbonMonoxideSensor.State,
```

```
--#             MethaneSensor.State,
--#             WaterFlowSensor.State,
--#             WaterLevelSensor.State,
--#             WaterLevelSensor.WaterLevelInterrupt,
--#             PumpSwitch.State,
--#             PumpSwitch.PumpSwitchInterrupt,
--#             Pump.State,
--#             AdaCalendar.ClockTime,
--#             BIT.PBITDone &
```

The log buffer annotation is identical to that for the system log with the exception that it is also derived from itself. This is because it is initialized to empty at start-up.

The state held by the active sensors is obviously derived from the relevant interrupts. Again we see only the abstract names for the interrupts.

```
--#          PumpSwitch.State from
--#             PumpSwitch.PumpSwitchInterrupt &

--#          WaterLevelSensor.State from
                WaterLevelSensor.WaterLevelInterrupt &
```

The suspension object `PBITDone` is derived from itself. Suspension objects are only ever derived from themselves at the partition level.

```
--#          BIT.PBITDone from * &
```

The clock is read to calculate the delay time for the tasks and is hence being used in a temporal capacity rather than actually affecting any data flow. Hence `null` is derived from the clock's own variable `ClockTime`.

```
--#          null from Ada.Real_Time.ClockTime;
```

4.4 Start-up

It is common for embedded systems to run tests at start-up and while the program is running. Such tests are known as built-in-tests (BIT). The power-up tests (PBIT) generally check the devices are initialised properly. The continuous tests (CBIT) perform more dynamic tests. We have encapsulated these tests in a thread (task) called BIT.

The pump switch and water level state are only updated when an interrupt occurs and hence at start-up the state is undefined for a short time until the power-up interrupts are detected. The function of PBIT is to monitor the interrupt driven devices and set a flag when both power-up interrupts have occurred.

The flag takes the form of a suspension object `PBITDone`. The pump controller task suspends on PBITDone and is released by the BIT task when the devices are ready.

```
-- Power-up BIT (PBIT)
loop
   PumpSwitchState := PumpSwitch.GetState;
   WaterLevelState := WaterLevelSensor.GetState;

   exit when PumpSwitchState /= PumpSwitch.Undefined and
             WaterLevelState /= WaterLevelSensor.Undefined;

   NextPeriod := NextPeriod + Period;
   delay until NextPeriod;
end loop;

-- PBIT complete.
Ada.Synchronous_Task_Control.Set_True (PBITDone);

-- Continuous BIT (CBIT)
loop
   <do CBIT>
   NextPeriod := NextPeriod + Period;
   delay until NextPeriod;
end loop;
```

4.5 Normal Operation

The threads of control drive the inputs to the outputs in accordance with the system's requirements. We have used a mixture of interrupt handlers, periodic and aperodic tasks to achieve the goal. The interrupt handlers were described in section 4.1.2. The remaining threads are now described.

4.5.1 Gas Monitors

The periodic `AirMonitor` task reads the air sensor and sets the light panel appropriately. The `CarbonMonoxideMonitor` and `MethaneMonitor` perform similar functions.

4.5.2 Pump Controller

The PumpController task drives the pump actuator and exhibits periodic and aperiodic behaviour.

```
BIT.WaitUntilPBITComplete;

NextPeriod := Ada.Real_Time.Clock;
loop
   NextPeriod := NextPeriod + Period;
   delay until NextPeriod;
   if <pump on logic> then
      Pump.SetState (To => Pump.On);
   else
      Pump.SetState (To => Pump.Off);
   end if;
end loop;
```

The task must first wait until released by the BIT task. When the task is released we are guaranteed that all the sensors are providing correct data. Once released the task acts as a periodic scheduler that reads the three sensors and determines if the pump should be on or off. This task runs with a period of 50ms to ensure that the spark is never given time to ignite the methane.

4.5.3 Logger

From the global and derives annotation on the task specification we can see that the Logger is responsible for transferring data from the LogBuffer to the SystemLog. The suspends property tells us that the task exhibits aperiodic behaviour and that the release criteria depends on `LogBuffer.State`.

```
task type LoggerTask
--# global in out LogBuffer.State;
--#           out SystemLog.State;
--#
--# derives SystemLog.State from LogBuffer.State &
--#         LogBuffer.State from *;
--# declare suspends => LogBuffer.State;
is
   pragma Priority (System.Priority'First + 1);
end LoggerTask;
```

In fact `LogBuffer.State` is a protected object with an entry. The logger task waits on this entry and when released gets the data and writes it to the system log.

```
loop
   LogBuffer.Read (TheData, DataMissed);
   SystemLog.Write (TheData, DataMissed);
end loop;
```

This task is given a low priority and will be pre-empted by more important events.

4.5.4 The Environment Task

The environment task is responsible for elaborating the program and starting the main procedure. The main procedure in this case is a null statement.

4.6 Scheduling

The table below shows all the threads, their type, period if applicable and priorities.

Thread	Type			Priority
	Interrupt Handler	Periodic	Aperiodic	
PumpSwitch.AutoInterruptHandler	●			Highest
PumpSwitch.OnInterruptHandler	●			Highest
PumpSwitch.OffInterruptHandler	●			Highest
WaterLevelSensor. WaterIsHighInterruptHandler	●			Highest
WaterLevelSensor. WaterIsLowInterruptHandler	●			Highest
PumpController		50ms	●	Highest
AirMonitor		100ms		High
CarbonMonoxideMonitor		100ms		High
MethaneMonitor		100ms		High
Logger			●	Low
BIT		100ms		Lowest
Environment thread				Lowest

A Mine Pump Requirements

A.1 Pump Operation

A.1.1 The pump has two modes manual and automatic.

A.1.2 In automatic mode the pump will pump water when the water level is too high.

A.1.3 In manual mode the operator shall be able to switch the pump on and off.

A.1.4 The MinePump shall NEVER operate when the methane level is too high. This applies to both manual and automatic modes.

A.1.5 The MinePump shall NEVER operate when the methane sensor is broken. This applies to both manual **and** automatic modes.

A.2 System State

A.2.1 The operator shall be informed of the amount of air in the shaft in relation to a threshold that is **deemed** safe.

A.2.2 The operator shall be informed of the amount of carbon monoxide in the shaft in relation to a threshold that is deemed safe.

A.2.3 The operator shall be informed of the amount of methane in the shaft in relation to a threshold that is deemed safe.

A.2.4 The operator shall be informed of the amount of water in the shaft in relation to the “high” point.

A.2.5 The operator shall be informed when the pump is pumping.

A.3 Diagnostics

A.3.1 The Operator shall be informed of the status of the following sensors: air, carbon monoxide and methane.

A.3.2 The Operator shall be informed when the pump is broken. The pump is broken when: the pump should be on and water is not flowing OR water is flowing and the pump should not be on.

A.3.3 A system log shall be kept that reports the times of the following events:

- Air level normal
- Air level too low
- Air sensor broken
- Carbon monoxide level normal
- Carbon monoxide level too high
- Carbon monoxide sensor broken
- Methane level normal
- Methane level too high
- Methane sensor broken
- Water level normal
- Water level too high
- Pump is on
- Pump is off
- Pump is broken

A.4 Hardware Interface

A.4.1 The pump switch and water level sensors are edge triggered interrupt devices. They do, however, provide an interrupt on power up.

A.4.2 The pump actuator device powers up in the off state.

A.4.3 The spark caused by operating the pump may ignite methane in 60ms.

A.4.4 The lights power up in the off state.

A.4.5 Sensors do not exhibit intermittent faults. They are either working or broken.

B Code

```
package TimeStamp
--# own protected ClockTime;
is

   type Time is private;

   NullTime : constant Time;

   function Clock return Time;
   --# global ClockTime;

private

   type Time is
     record
        I : Integer;
     end record;

   NullTime : constant Time := Time'(I => 0);

end TimeStamp;
----------------------------------------------------------------------------


with TimeStamp;
with System;
--# inherit TimeStamp,
--#         System;
package AirSensor
--# own           in RawSensor;
--#     protected in State : SensorPT (Priority => System.Interrupt_Priority'Last,
--#                                    Protects => RawSensor);
is

   type Value is range 0 .. 255;

   MinimumSafeValue : constant Value := 100;

   procedure GetReading (TheReading : out Value;
                         TheTime    : out TimeStamp.Time);
   --# global in State;
   --#        in TimeStamp.ClockTime;
   --# derives TheReading from State &
   --#         TheTime    from TimeStamp.ClockTime;

private

   protected type SensorPT is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

       procedure ProtectedGetReading (TheReading : out Value;
                                      TheTime    : out TimeStamp.Time);
       --# global in SensorPT;
       --#        in TimeStamp.ClockTime;
       --# derives TheReading from SensorPT &
       --#         TheTime    from TimeStamp.ClockTime;

   end SensorPT;

end AirSensor;
----------------------------------------------------------------------------
```

```
with TimeStamp;
with System;
--# inherit TimeStamp,
--#         System;
package CarbonMonoxideSensor
--# own           in RawSensor;
--#     protected in State : SensorPT (Priority => System.Interrupt_Priority'Last,
--#                                   Protects => RawSensor);
is

   type Value is range 0 .. 255;

   MaximumSafeValue : constant Value := 100;

   procedure GetReading (TheReading : out Value;
                         TheTime    : out TimeStamp.Time);
   --# global in State;
   --#        in TimeStamp.ClockTime;
   --# derives TheReading from State &
   --#         TheTime    from TimeStamp.ClockTime;

private

   protected type SensorPT is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

       procedure ProtectedGetReading (TheReading : out Value;
                                      TheTime    : out TimeStamp.Time);
       --# global in SensorPT;
       --#        in TimeStamp.ClockTime;
       --# derives TheReading from SensorPT &
       --#         TheTime    from TimeStamp.ClockTime;

   end SensorPT;

end CarbonMonoxideSensor;
-----------------------------------------------------------------------------
```

```
with TimeStamp;
with System;
--# inherit TimeStamp,
--#         System;
package MethaneSensor
--# own           in RawSensor;
--#     protected in State : SensorPT (Priority => System.Interrupt_Priority'Last,
--#                                   Protects => RawSensor);
is

   type Value is range 0 .. 255;
   pragma Atomic (Value);

   MaximumSafeValue : constant Value := 100;

   procedure GetReading (TheReading : out Value;
                         TheTime    : out TimeStamp.Time);
   --# global in State;
   --#        in TimeStamp.ClockTime;
   --# derives TheReading from State &
   --#         TheTime    from TimeStamp.ClockTime;
```

```
private

   protected type SensorPT is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

       procedure ProtectedGetReading (TheReading : out Value;
                                      TheTime    : out TimeStamp.Time);
       --# global in SensorPT;
       --#        in TimeStamp.ClockTime;
       --# derives TheReading from SensorPT &
       --#         TheTime    from TimeStamp.ClockTime;

   end SensorPT;

end MethaneSensor;
----------------------------------------------------------------------------



with TimeStamp;
with System;
--# inherit TimeStamp,
--#         System;
package WaterFlowSensor
--# own           in RawSensor;
--#     protected in State : SensorPT (Priority => System.Interrupt_Priority'Last,
--#                                    Protects => RawSensor);
is

   type Value is (Flowing, NotFlowing);

   procedure GetReading (TheReading : out Value;
                         TheTime    : out TimeStamp.Time);
   --# global in State;
   --#        in TimeStamp.ClockTime;
   --# derives TheReading from State &
   --#         TheTime    from TimeStamp.ClockTime;

private

   protected type SensorPT is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

       procedure ProtectedGetReading (TheReading : out Value;
                                      TheTime    : out TimeStamp.Time);
       --# global in SensorPT;
       --#        in TimeStamp.ClockTime;
       --# derives TheReading from SensorPT &
       --#         TheTime    from TimeStamp.ClockTime;

   end SensorPT;

end WaterFlowSensor;
----------------------------------------------------------------------------
```

```
with TimeStamp;
with System;
--# inherit TimeStamp,
--#         System,
--#         LightPanel,
--#         LogBuffer;
package WaterLevelSensor
--# own protected State :
--#            SensorPT
--#               (Priority => System.Interrupt_Priority'Last,
--#                Interrupt => (WaterIsHighInterruptHandler => WaterLevelInterrupt,
--#                              WaterIsLowInterruptHandler  => WaterLevelInterrupt));
is

   type Value is (Undefined, Low, High);

   function GetState return Value;
   --# global in State;

private

   protected type SensorPT
   is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

      procedure WaterIsHighInterruptHandler;
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out SensorPT;
      --#           out LightPanel.WaterLevelLight;
      --# derives SensorPT,
      --#         LightPanel.WaterLevelLight from  &
      --#         LogBuffer.State            from *,
      --#                                         TimeStamp.ClockTime;
      pragma Attach_Handler (WaterIsHighInterruptHandler, 1);

      procedure WaterIsLowInterruptHandler;
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out SensorPT;
      --#           out LightPanel.WaterLevelLight;
      --# derives SensorPT,
      --#         LightPanel.WaterLevelLight from  &
      --#         LogBuffer.State            from *,
      --#                                         TimeStamp.ClockTime;
      pragma Attach_Handler (WaterIsLowInterruptHandler, 1);

      function ProtectedGetState return Value;
      --# global in SensorPT;

   private
      CurrentState : Value := Undefined;
   end SensorPT;

end WaterLevelSensor;
-----------------------------------------------------------------------------
```

```
with TimeStamp;
with LightPanel;
package body WaterLevelSensor
is

   State : SensorPT;

   protected body SensorPT is

      procedure WaterIsHighInterruptHandler
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out CurrentState;
      --#           out LightPanel.WaterLevelLight;
      --# derives CurrentState,
      --#         LightPanel.WaterLevelLight from  &
      --#         LogBuffer.State            from *, TimeStamp.ClockTime;
      is
         TheTime : TimeStamp.Time;
      begin
         CurrentState := High;
         TheTime := TimeStamp.Clock;
         LightPanel.SwitchWaterLevelLight (IsHigh => True,
                                           TheTimeStamp => TheTime);
      end WaterIsHighInterruptHandler;

      procedure WaterIsLowInterruptHandler
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out CurrentState;
      --#           out LightPanel.WaterLevelLight;
      --# derives CurrentState,
      --#         LightPanel.WaterLevelLight from  &
      --#         LogBuffer.State            from *, TimeStamp.ClockTime;
      is
         TheTime : TimeStamp.Time;
      begin
         CurrentState := Low;
         TheTime := TimeStamp.Clock;
         LightPanel.SwitchWaterLevelLight (IsHigh => False,
                                           TheTimeStamp => TheTime);
      end WaterIsLowInterruptHandler;


      function ProtectedGetState return Value
      --# global in CurrentState;
      is
      begin
         return CurrentState;
      end ProtectedGetState;

   end SensorPT;


   function GetState return Value  is
   begin
      return State.ProtectedGetState;
   end GetState;

end WaterLevelSensor;
----------------------------------------------------------------------------
```

```
with System;
--# inherit TimeStamp,
--#         System,
--#         LogBuffer;
package PumpSwitch
--# own protected State :
--#        PumpSwitchPT
--#           (Priority => System.Interrupt_Priority'Last,
--#            interrupt);
is

   type Value is (Undefined, Auto, On, Off);

   function GetState return Value;
   --# global State;

private

   protected type PumpSwitchPT
   is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

      procedure AutoInterruptHandler;
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out PumpSwitchPT;
      --# derives PumpSwitchPT    from  &
      --#         LogBuffer.State from *,
      --#                              TimeStamp.ClockTime;
      pragma Attach_Handler (AutoInterruptHandler, 3);

      procedure OnInterruptHandler;
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out PumpSwitchPT;
      --# derives PumpSwitchPT    from  &
      --#         LogBuffer.State from *,
      --#                              TimeStamp.ClockTime;
      pragma Attach_Handler (OnInterruptHandler, 4);

      procedure OffInterruptHandler;
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out PumpSwitchPT;
      --# derives PumpSwitchPT    from  &
      --#         LogBuffer.State from *,
      --#                              TimeStamp.ClockTime;
      pragma Attach_Handler (OffInterruptHandler, 5);

      function ProtectedGetState return Value;
      --# global in PumpSwitchPT;

   private
      CurrentState : Value := Undefined;
   end PumpSwitchPT;

end PumpSwitch;
----------------------------------------------------------------------------
```

```
with TimeStamp;
with LogBuffer;
package body PumpSwitch is

   State : PumpSwitchPT;

   protected body PumpSwitchPT
      is

      procedure AutoInterruptHandler
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out CurrentState;
      --# derives CurrentState    from  &
      --#         LogBuffer.State from *, TimeStamp.ClockTime;
      is
         TheTime : TimeStamp.Time;
      begin
         CurrentState := Auto;
         TheTime := TimeStamp.Clock;
         LogBuffer.Write (Data => LogBuffer.BufferEntry'
                             (TheDevice => LogBuffer.PumpSwitch,
                              TheState  => LogBuffer.IsAuto,
                              TheTimeStamp => TheTime));
      end AutoInterruptHandler;

      procedure OnInterruptHandler
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out CurrentState;
      --# derives CurrentState    from  &
      --#         LogBuffer.State from *, TimeStamp.ClockTime;
      is
         TheTime : TimeStamp.Time;
      begin
         CurrentState := On;
         TheTime := TimeStamp.Clock;
         LogBuffer.Write (Data => LogBuffer.BufferEntry'
                             (TheDevice => LogBuffer.PumpSwitch,
                              TheState  => LogBuffer.IsOn,
                              TheTimeStamp => TheTime));
      end OnInterruptHandler;

      procedure OffInterruptHandler
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out CurrentState;
      --# derives CurrentState    from  &
      --#         LogBuffer.State from *, TimeStamp.ClockTime;
      is
         TheTime : TimeStamp.Time;
      begin
         CurrentState := Off;
         TheTime := TimeStamp.Clock;
         LogBuffer.Write (Data => LogBuffer.BufferEntry'
                             (TheDevice => LogBuffer.PumpSwitch,
                              TheState  => LogBuffer.IsOff,
                              TheTimeStamp => TheTime));
      end OffInterruptHandler;

      function ProtectedGetState return Value
      --# global in CurrentState;
      is
      begin
         return CurrentState;
      end ProtectedGetState;
```

```
   end PumpSwitchPT;

   function GetState return Value is
   begin
      return State.ProtectedGetState;
   end GetState;

end PumpSwitch;
----------------------------------------------------------------------------



with System;
--# inherit TimeStamp,
--#         System,
--#         LogBuffer;
package Pump
--# own protected out Actuator :
--#                         ActuatorPT (Priority => System.Interrupt_Priority'Last);
--#               in  State;
is

   type PumpState is (On, Off);

   procedure SetState (To : in PumpState);
   --# global in     TimeStamp.ClockTime;
   --#        in out LogBuffer.State;
   --#           out Actuator;
   --# derives Actuator        from To &
   --#         LogBuffer.State from *,
   --#                              To,
   --#                              TimeStamp.ClockTime;

   function IsOn return Boolean;
   --# global in State;

private

   protected type ActuatorPT is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

      procedure protectedSetState (To : in PumpState);
      --# global in     TimeStamp.ClockTime;
      --#        in out LogBuffer.State;
      --#           out Actuator;
      --# derives Actuator        from To &
      --#         LogBuffer.State from *,
      --#                              To,
      --#                              TimeStamp.ClockTime;

   end ActuatorPT;

end Pump;
----------------------------------------------------------------------------
```

```
with TimeStamp;
--# inherit TimeStamp,
--#         LogBuffer;
package LightPanel
--# own out       AirLight;
--#     out       CarbonMonoxideLight;
--#     out       MethaneLight;
--#     out       PumpHealthLight;
--#     out       PumpPumpingLight;
--#     protected WaterLevelLight;
is

   type Level is (Normal, Dangerous, Unknown);

   type HealthStatus is (Healthy, Unhealthy, Unavailable);

   type PumpingStatus is (Pumping, NotPumping, PumpingUnknown);

   procedure SwitchAirLight (To           : in Level;
                             TheTimeStamp : in TimeStamp.Time);
   --# global in out LogBuffer.State;
   --#            out AirLight;
   --# derives AirLight        from To &
   --#         LogBuffer.State from *,
   --#                              To,
   --#                              TheTimeStamp;

   procedure SwitchCarbonMonoxideLight (To           : in Level;
                                        TheTimeStamp : in TimeStamp.Time);
   --# global in out LogBuffer.State;
   --#            out CarbonMonoxideLight;
   --# derives LogBuffer.State     from *,
   --#                                  To,
   --#                                  TheTimeStamp &
   --#         CarbonMonoxideLight from To;

   procedure SwitchMethaneLight (To           : in Level;
                                 TheTimeStamp : in TimeStamp.Time);
   --# global in out LogBuffer.State;
   --#            out MethaneLight;
   --# derives LogBuffer.State from *,
   --#                              To,
   --#                              TheTimeStamp &
   --#         MethaneLight    from To;

   procedure SwitchWaterLevelLight (IsHigh       : in Boolean;
                                    TheTimeStamp : in TimeStamp.Time);
   --# global in out LogBuffer.State;
   --#            out WaterLevelLight;
   --# derives LogBuffer.State from *,
   --#                              TheTimeStamp,
   --#                              IsHigh &
   --#         WaterLevelLight from IsHigh;

   procedure SwitchPumpHealthLight (To           : in HealthStatus;
                                    TheTimeStamp : in TimeStamp.Time);
   --# global in out LogBuffer.State;
   --#            out PumpHealthLight;
   --# derives LogBuffer.State from *,
   --#                              To,
   --#                              TheTimeStamp &
   --#         PumpHealthLight from To;
```

```
   procedure SwitchPumpPumpingLight (To           : in PumpingStatus;
                                     TheTimeStamp : in TimeStamp.Time);
   --# global in out LogBuffer.State;
   --#           out PumpPumpingLight;
   --# derives LogBuffer.State  from *,
   --#                              To,
   --#                              TheTimeStamp &
   --#         PumpPumpingLight from To;

end LightPanel;
----------------------------------------------------------------------------


with TimeStamp;
with System;
--# inherit TimeStamp,
--#         System;
package LogBuffer
--# own protected State : Log (Priority => System.Interrupt_Priority'Last,
--#                            Suspendable);
is

   type Device is
     (UnknownDevice,
      AirSensor,
      CarbonMonoxideSensor,
      MethaneSensor,
      WaterLevelSensor,
      WaterFlowSensor,
      Lights,
      Pump,
      PumpSwitch);

   type DeviceState is
     (UnknownState,
      IsOn,
      IsOff,
      IsAuto,
      IsDangerous,
      IsBroken,
      IsOk,
      IsHigh);

   type BufferEntry is
      record
         TheDevice : Device;
         TheState  : DeviceState;
         TheTimeStamp : TimeStamp.Time;
      end record;

   NullBufferEntry : constant BufferEntry :=
     BufferEntry'(TheDevice => UnknownDevice,
                  TheState  => UnknownState,
                  TheTimeStamp => TimeStamp.NullTime);


   procedure Write (Data : in BufferEntry);
   --# global in out State;
   --# derives State from *,
   --#                    Data;
```

```
   procedure Read (Data       : out BufferEntry;
                   DataMissed : out Boolean);
   --# global in out State;
   --# derives State,
   --#         Data,
   --#         DataMissed from State;
   --# declare suspends => State;

private

   MaxEntries : constant := 1024;
   type Index is mod MaxEntries;

   type Buffer is array (Index) of BufferEntry;

   EmptyBuffer : constant Buffer := Buffer'(others => NullBufferEntry);

   protected type Log
      is

      pragma Interrupt_Priority (System.Interrupt_Priority'Last);

      procedure ProtectedWrite (Data : in BufferEntry);
      --# global in out Log;
      --# derives Log from *,
      --#                  Data;

      entry ProtectedRead (Data       : out BufferEntry;
                           DataMissed : out Boolean);
      --# global in out Log;
      --# derives Data,
      --#         DataMissed,
      --#         Log        from Log;

   private
      TheBuffer  : Buffer  := EmptyBuffer;
      ReadIndex  : Index   := Index'First;
      WriteIndex : Index   := Index'First;
      HasEntries : Boolean := False;
      DataOverwrittenSinceLastRead : Boolean := False;
   end Log;

end LogBuffer;
-----------------------------------------------------------------------------


package body LogBuffer
is

   State : Log;

   protected body Log
      is

      procedure ProtectedWrite (Data : in BufferEntry)
      --# global in out TheBuffer;
      --#        in out WriteIndex;
      --#        in out ReadIndex;
      --#        in out DataOverwrittenSinceLastRead;
      --#        in out HasEntries;
      --# derives ReadIndex,
      --#         DataOverwrittenSinceLastRead from *,
      --#                                           ReadIndex,
      --#                                           WriteIndex,
      --#                                           HasEntries &
```

```
      --#         WriteIndex                   from * &
      --#         HasEntries                   from  &
      --#         TheBuffer                    from *,
      --#                                           WriteIndex,
      --#                                           Data;
      is
      begin
         TheBuffer (WriteIndex) := Data;

         if HasEntries and
           WriteIndex = ReadIndex
         then
            -- We have just overwritten the oldest entry
            -- Move the read pointer to the next oldest entry and mark that
            -- data has been overwritten.
            ReadIndex := ReadIndex + 1;
            DataOverwrittenSinceLastRead := True;
         end if;
         WriteIndex := WriteIndex + 1;
         HasEntries := True;
      end ProtectedWrite;


      entry ProtectedRead (Data       : out BufferEntry;
                           DataMissed : out Boolean) when HasEntries
      --# global in     TheBuffer;
      --#        in     WriteIndex;
      --#        in out ReadIndex;
      --#        in out DataOverwrittenSinceLastRead;
      --#           out HasEntries;
      --# derives ReadIndex                    from * &
      --#         DataOverwrittenSinceLastRead from  &
      --#         HasEntries                   from ReadIndex,
      --#                                           WriteIndex &
      --#         Data                         from ReadIndex,
      --#                                           TheBuffer &
      --#         DataMissed                   from DataOverwrittenSinceLastRead;
      is
      begin
         Data       := TheBuffer (ReadIndex);
         ReadIndex  := ReadIndex + 1;
         HasEntries := ReadIndex /= WriteIndex;
         DataMissed := DataOverwrittenSinceLastRead;
         DataOverwrittenSinceLastRead := False;
      end ProtectedRead;

   end Log;


   procedure Write (Data : in BufferEntry)
   is
   begin
      State.ProtectedWrite (Data);
   end Write;

   procedure Read (Data       : out BufferEntry;
                   DataMissed : out Boolean)
   is
   begin
      State.ProtectedRead (Data, DataMissed);
   end Read;

end LogBuffer;
----------------------------------------------------------------------------
```

```
with LogBuffer;
--# inherit LogBuffer;
package SystemLog
--# own out State;
is

   procedure Write (Data       : in LogBuffer.BufferEntry;
                    DataMissed : in Boolean);
   --# global out State;
   --# derives State from Data,
   --#                     DataMissed;
   --# declare delay;

end SystemLog;
----------------------------------------------------------------------------
```

```
with AirMonitor;
with CarbonMonoxideMonitor;
with MethaneMonitor;
with WaterLevelSensor;
with PumpController;
with BIT;
with PumpSwitch;
with Logger;

--# inherit TimeStamp,
--#         Ada.Real_Time,
--#         AirMonitor,
--#         AirSensor,
--#         CarbonMonoxideMonitor,
--#         CarbonMonoxideSensor,
--#         MethaneMonitor,
--#         MethaneSensor,
--#         WaterFlowSensor,
--#         WaterLevelSensor,
--#         LightPanel,
--#         Pump,
--#         PumpController,
--#         BIT,
--#         PumpSwitch,
--#         SystemLog,
--#         LogBuffer,
--#         Logger;

--# main_program;

-- Partition wide annotation.

--# global in      AirSensor.State;
--#        in      CarbonMonoxideSensor.State;
--#        in      MethaneSensor.State;
--#        in      WaterFlowSensor.State;
--#        in      Pump.State;
--#        in      TimeStamp.ClockTime;
--#        in      Ada.Real_Time.ClockTime;
--#        in      WaterLevelSensor.WaterLevelInterrupt;
--#        in out WaterLevelSensor.State;
--#        in out PumpSwitch.State;
--#        in out LogBuffer.State;
--#        in out BIT.PBITDone;
--#           out SystemLog.State;
--#           out LightPanel.AirLight;
```

```
--#            out LightPanel.CarbonMonoxideLight;
--#            out LightPanel.MethaneLight;
--#            out LightPanel.WaterLevelLight;
--#            out LightPanel.PumpHealthLight;
--#            out LightPanel.PumpPumpingLight;
--#            out Pump.Actuator;

--# derives SystemLog.State                from LogBuffer.State,
--#                                              AirSensor.State,
--#                                              CarbonMonoxideSensor.State,
--#                                              MethaneSensor.State,
--#                                              WaterFlowSensor.State,
--#                                              WaterLevelSensor.State,
--#                                              WaterLevelSensor.WaterLevelInterrupt,
--#                                              PumpSwitch.State,
--#                                              Pump.State,
--#                                              TimeStamp.ClockTime,
--#                                              BIT.PBITDone &

--#         LogBuffer.State                from *,
--#                                              AirSensor.State,
--#                                              CarbonMonoxideSensor.State,
--#                                              MethaneSensor.State,
--#                                              WaterFlowSensor.State,
--#                                              WaterLevelSensor.State,
--#                                              WaterLevelSensor.WaterLevelInterrupt,
--#                                              PumpSwitch.State,
--#                                              Pump.State,
--#                                              TimeStamp.ClockTime,
--#                                              BIT.PBITDone &

--#         LightPanel.AirLight            from AirSensor.State &

--#         LightPanel.CarbonMonoxideLight from CarbonMonoxideSensor.State &

--#         LightPanel.MethaneLight        from MethaneSensor.State &

--#         LightPanel.WaterLevelLight     from WaterLevelSensor.WaterLevelInterrupt &

--#         LightPanel.PumpHealthLight     from WaterFlowSensor.State,
--#                                              Pump.State &

--#         LightPanel.PumpPumpingLight    from WaterFlowSensor.State &

--#         Pump.Actuator                  from MethaneSensor.State,
--#                                              WaterLevelSensor.State,
--#                                              WaterLevelSensor.WaterLevelInterrupt,
--#                                              PumpSwitch.State,
--#                                              BIT.PBITDone &

--#         PumpSwitch.State               from PumpSwitch.State &
--#         WaterLevelSensor.State         from WaterLevelSensor.WaterLevelInterrupt &

--#         BIT.PBITDone                   from * &
--#         null                           from Ada.Real_Time.ClockTime;

procedure Main
--# derives ;
is
   pragma Priority (1);
begin
   null;
end Main;
----------------------------------------------------------------------------
```

```
with System;
--# inherit Ada.Real_Time,
--#         TimeStamp,
--#         AirSensor,
--#         LightPanel,
--#         System,
--#         LogBuffer;
package AirMonitor
--# own task TheMonitor : Monitor;
is

private

   task type Monitor
   --# global in     AirSensor.State;
   --#        in     Ada.Real_Time.ClockTime;
   --#        in     TimeStamp.ClockTime;
   --#        in out LogBuffer.State;
   --#           out LightPanel.AirLight;
   --# derives LightPanel.AirLight from AirSensor.State &
   --#         LogBuffer.State     from *,
   --#                                  AirSensor.State,
   --#                                  TimeStamp.ClockTime &
   --#         null                from Ada.Real_Time.ClockTime;
   is
      pragma Priority (System.Priority'Last);
   end Monitor;

end AirMonitor;
----------------------------------------------------------------------------



with Ada.Real_Time;
with TimeStamp;
with AirSensor;
with LightPanel;

use type AirSensor.Value;
use type Ada.Real_Time.Time;

package body AirMonitor
is

   TheMonitor : Monitor;

   task body Monitor
   is

      Period : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds (100);

      TheAirReading   : AirSensor.Value;
      TheTime         : TimeStamp.Time;
      NextPeriod      : Ada.Real_Time.Time;
      NewLightSetting : LightPanel.Level;
   begin
      NextPeriod := Ada.Real_Time.Clock;
      loop
         AirSensor.GetReading (TheReading => TheAirReading,
                               TheTime    => TheTime);

         if TheAirReading'Valid then
            if TheAirReading >= AirSensor.MinimumSafeValue then
               NewLightSetting := LightPanel.Normal;
            else
```

```
            NewLightSetting := LightPanel.Dangerous;
         end if;
      else
         NewLightSetting := LightPanel.Unknown;
      end if;

      LightPanel.SwitchAirLight (To => NewLightSetting,
                                 TheTimeStamp => TheTime);

      NextPeriod := NextPeriod + Period;
      delay until NextPeriod;

   end loop;
end Monitor;

end AirMonitor;
------------------------------------------------------------------------------
```

```
with System;

--# inherit Ada.Real_Time,
--#         TimeStamp,
--#         CarbonMonoxideSensor,
--#         LightPanel,
--#         System,
--#         LogBuffer;

package CarbonMonoxideMonitor
--# own task TheMonitor : Monitor;
is

private

   task type Monitor
     --# global in      CarbonMonoxideSensor.State;
     --#        in      Ada.Real_Time.ClockTime;
     --#        in      TimeStamp.ClockTime;
     --#        in out LogBuffer.State;
     --#           out LightPanel.CarbonMonoxideLight;
     --#
     --# derives LightPanel.CarbonMonoxideLight from CarbonMonoxideSensor.State &
     --#         LogBuffer.State                from *,
     --#                                             CarbonMonoxideSensor.State,
     --#                                             TimeStamp.ClockTime &
     --#         null                           from Ada.Real_Time.ClockTime;
   is
      pragma Priority (System.Priority'Last);
   end Monitor;

end CarbonMonoxideMonitor;
------------------------------------------------------------------------------
```

```
with Ada.Real_Time;
with TimeStamp;
with CarbonMonoxideSensor;
with LightPanel;

use type CarbonMonoxideSensor.Value;
use type Ada.Real_Time.Time;

package body CarbonMonoxideMonitor
is

   TheMonitor : Monitor;

   task body Monitor
   is

      Period : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds (100);

      TheCOReading  : CarbonMonoxideSensor.Value;
      TheTime       : TimeStamp.Time;
      NextPeriod    : Ada.Real_Time.Time;
      NewLightValue : LightPanel.Level;

   begin
      NextPeriod := Ada.Real_Time.Clock;
      loop
         CarbonMonoxideSensor.GetReading (TheReading => TheCOReading,
                                          TheTime    => TheTime);

         if TheCOReading'Valid then
            if TheCOReading <= CarbonMonoxideSensor.MaximumSafeValue then
               NewLightValue := LightPanel.Normal;
            else
               NewLightValue := LightPanel.Dangerous;
            end if;
         else
            NewLightValue := LightPanel.Unknown;
         end if;

         LightPanel.SwitchCarbonMonoxideLight
           (To => NewLightValue,
            TheTimeStamp => TheTime);

         NextPeriod := NextPeriod + Period;
         delay until NextPeriod;

      end loop;
   end Monitor;

end CarbonMonoxideMonitor;
------------------------------------------------------------------------------
```

```
with System;

--# inherit Ada.Real_Time,
--#         TimeStamp,
--#         MethaneSensor,
--#         LightPanel,
--#         System,
--#         LogBuffer;

package MethaneMonitor
--# own task TheMonitor : Monitor;
is

private

   task type Monitor
     --# global in     MethaneSensor.State;
     --#        in     Ada.Real_Time.ClockTime;
     --#        in     TimeStamp.ClockTime;
     --#        in out LogBuffer.State;
     --#           out LightPanel.MethaneLight;
     --#
     --# derives LightPanel.MethaneLight from MethaneSensor.State &
     --#         LogBuffer.State         from *,
     --#                                      MethaneSensor.State,
     --#                                      TimeStamp.ClockTime &
     --#         null                    from Ada.Real_Time.ClockTime;
      is
      pragma Priority (System.Priority'Last);

   end Monitor;

end MethaneMonitor;
-----------------------------------------------------------------------------



with Ada.Real_Time;
with TimeStamp;
with MethaneSensor;
with LightPanel;

use type MethaneSensor.Value;
use type Ada.Real_Time.Time;

package body MethaneMonitor
is

   TheMonitor : Monitor;

   task body Monitor
   is

      Period : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds (100);

      TheMethaneReading : MethaneSensor.Value;
      TheTime           : TimeStamp.Time;
      NextPeriod        : Ada.Real_Time.Time;
      NewLightValue     : LightPanel.Level;

   begin
      NextPeriod := Ada.Real_Time.Clock;
      loop
         MethaneSensor.GetReading (TheReading => TheMethaneReading,
                                   TheTime    => TheTime);
```

```
         if TheMethaneReading'Valid then
            if TheMethaneReading <= MethaneSensor.MaximumSafeValue then
               NewLightValue := LightPanel.Normal;
            else
               NewLightValue := LightPanel.Dangerous;
            end if;
         else
            NewLightValue := LightPanel.Unknown;
         end if;

         LightPanel.SwitchMethaneLight
           (To => NewLightValue,
            TheTimeStamp => TheTime);

         NextPeriod := NextPeriod + Period;
         delay until NextPeriod;

      end loop;
   end Monitor;

end MethaneMonitor;
------------------------------------------------------------------------------



with System;
--# inherit Ada.Real_Time,
--#         Ada.Synchronous_Task_Control,
--#         TimeStamp,
--#         WaterFlowSensor,
--#         Pump,
--#         PumpSwitch,
--#         LightPanel,
--#         System,
--#         LogBuffer,
--#         WaterLevelSensor;

package BIT
--# own task TheMonitor : Monitor;
--#     protected PBITDone (suspendable);
is

   procedure WaitUntilPBITComplete;
   --# global PBITDone;
   --# derives PBITDone from ;
   --# declare suspends => PBITDone;

private

   task type Monitor
   --# global in      WaterFlowSensor.State;
   --#        in      Pump.State;
   --#        in      TimeStamp.ClockTime;
   --#        in      Ada.Real_Time.ClockTime;
   --#        in      PumpSwitch.State;
   --#        in      WaterLevelSensor.State;
   --#        in out  LogBuffer.State;
   --#           out  PBITDone;
   --#           out  LightPanel.PumpHealthLight;
   --#           out  LightPanel.PumpPumpingLight;
   --# derives LightPanel.PumpHealthLight  from Pump.State,
   --#                                          WaterFlowSensor.State &
   --#         LightPanel.PumpPumpingLight from WaterFlowSensor.State &
   --#         LogBuffer.State             from *,
   --#                                          Pump.State,
   --#                                          WaterFlowSensor.State,
```

```
   --#                                          TimeStamp.ClockTime &
   --#         PBITDone                    from &
   --#         null                        from Ada.Real_Time.ClockTime,
   --#                                          PumpSwitch.State,
   --#                                          WaterLevelSensor.State;
   is
      pragma Priority (System.Priority'First);
   end Monitor;

end BIT;
--------------------------------------------------------------------------------
```

```
with Ada.Real_Time;
with Ada.Synchronous_Task_Control;
with WaterFlowSensor;
with LightPanel;
with Pump;
with PumpSwitch;
with WaterLevelSensor;
with TimeStamp;

use type Ada.Real_Time.Time;
use type WaterFlowSensor.Value;
use type WaterLevelSensor.Value;
use type PumpSwitch.Value;

package body BIT
is

   PBITDone : Ada.Synchronous_Task_Control.Suspension_Object;

   TheMonitor : Monitor;

   task body Monitor
   is

      Period : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds (100);

      PumpIsOn          : Boolean;
      WaterFlowReading : WaterFlowSensor.Value;
      PumpSwitchState   : PumpSwitch.Value;
      WaterLevelState   : WaterLevelSensor.Value;
      TheTime           : TimeStamp.Time;
      NextPeriod        : Ada.Real_Time.Time;
      NewPumpingStatus : LightPanel.PumpingStatus;
      NewHealthStatus   : LightPanel.HealthStatus;

   begin
      NextPeriod := Ada.Real_Time.Clock;

      --  Power-up BIT (PBIT)
      loop
         PumpSwitchState := PumpSwitch.GetState;
         WaterLevelState := WaterLevelSensor.GetState;

         exit when PumpSwitchState /= PumpSwitch.Undefined and
           WaterLevelState /= WaterLevelSensor.Undefined;

         NextPeriod := NextPeriod + Period;
         delay until NextPeriod;
      end loop;

      --  PBIT complete.
      Ada.Synchronous_Task_Control.Set_True (PBITDone);
```

```
      --  Continuous BIT (CBIT)
      loop
         NextPeriod := NextPeriod + Period;
         delay until NextPeriod;

         WaterFlowSensor.GetReading (TheReading => WaterFlowReading,
                                     TheTime    => TheTime);

         if WaterFlowReading'Valid then

            if WaterFlowReading = WaterFlowSensor.Flowing then
               NewPumpingStatus := LightPanel.Pumping;
            else
               NewPumpingStatus := LightPanel.NotPumping;
            end if;

            PumpIsOn := Pump.IsOn;

            if PumpIsOn xor
              WaterFlowReading = WaterFlowSensor.Flowing then
               NewHealthStatus := LightPanel.Unhealthy;
            else
               NewHealthStatus := LightPanel.Healthy;
            end if;
         else
            NewPumpingStatus := LightPanel.PumpingUnknown;
            NewHealthStatus := LightPanel.Unavailable;
         end if;

         LightPanel.SwitchPumpPumpingLight (To => NewPumpingStatus,
                                            TheTimeStamp => TheTime);

         LightPanel.SwitchPumpHealthLight (To => NewHealthStatus,
                                           TheTimeStamp => TheTime);
      end loop;
   end Monitor;

   procedure WaitUntilPBITComplete
   is
   begin
      Ada.Synchronous_Task_Control.Suspend_Until_True (PBITDone);
   end WaitUntilPBITComplete;

end BIT;
-----------------------------------------------------------------------------
```

```
with System;

--# inherit Ada.Real_Time,
--#         TimeStamp,
--#         BIT,
--#         MethaneSensor,
--#         WaterLevelSensor,
--#         Pump,
--#         PumpSwitch,
--#         LogBuffer,
--#         System,
--#         LightPanel;

package PumpController
--# own task TheController : Controller;
is

private

   task type Controller
   --# global in      WaterLevelSensor.State;
   --#        in      MethaneSensor.State;
   --#        in      PumpSwitch.State;
   --#        in      Ada.Real_Time.ClockTime;
   --#        in      TimeStamp.ClockTime;
   --#        in out  LogBuffer.State;
   --#           out  BIT.PBITDone;
   --#           out  Pump.Actuator;
   --#
   --# derives Pump.Actuator   from WaterLevelSensor.State,
   --#                              MethaneSensor.State,
   --#                              PumpSwitch.State &
   --#         LogBuffer.State from *,
   --#                              WaterLevelSensor.State,
   --#                              MethaneSensor.State,
   --#                              PumpSwitch.State,
   --#                              TimeStamp.ClockTime &
   --#         BIT.PBITDone    from &
   --#         null            from Ada.Real_Time.ClockTime;
   --# declare suspends => BIT.PBITDone;
   is
      pragma Interrupt_Priority (System.Interrupt_Priority'Last);
   end Controller;

end PumpController;
----------------------------------------------------------------------------
```

```
with Ada.Real_Time;
with TimeStamp;
with BIT;
with MethaneSensor;
with Pump;
with PumpSwitch;
with WaterLevelSensor;

use type Ada.Real_Time.Time;
use type MethaneSensor.Value;
use type PumpSwitch.Value;
use type WaterLevelSensor.Value;

package body PumpController
is
   TheController : Controller;

   task body Controller is

      Period : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds (50);

      TheMethaneReading : MethaneSensor.Value;
      PumpSwitchState   : PumpSwitch.Value;
      WaterLevelState   : WaterLevelSensor.Value;
      UnusedTimeStamp   : TimeStamp.Time;
      NextPeriod        : Ada.Real_Time.Time;

   begin
      BIT.WaitUntilPBITComplete;

      NextPeriod := Ada.Real_Time.Clock;
      loop
         NextPeriod := NextPeriod + Period;
         delay until NextPeriod;

         MethaneSensor.GetReading (TheReading => TheMethaneReading,
                                   TheTime    => UnusedTimeStamp);

         if TheMethaneReading'Valid and then
           TheMethaneReading <= MethaneSensor.MaximumSafeValue then

            PumpSwitchState := PumpSwitch.GetState;
            WaterLevelState := WaterLevelSensor.GetState;

            if PumpSwitchState = PumpSwitch.Auto then
               if WaterLevelState = WaterLevelSensor.High then
                  Pump.SetState (To => Pump.On);
               else
                  Pump.SetState (To => Pump.Off);
               end if;
            elsif PumpSwitchState = PumpSwitch.On then
               Pump.SetState (To => Pump.On);
            else
               Pump.SetState (To => Pump.Off);
            end if;

         else
            -- Methane sensor is broken or the methane level not safe.
            -- Switch off the pump.
            Pump.SetState (To => Pump.Off);
         end if;
      end loop;
   end Controller;

end PumpController;
--------------------------------------------------------------------------
```

```
with System;

--# inherit LogBuffer,
--#         SystemLog,
--#         System;

package Logger
--# own task TheLogger : LoggerTask;
is

private

   task type LoggerTask
     --# global in out LogBuffer.State;
     --#           out SystemLog.State;
     --#
     --# derives SystemLog.State from LogBuffer.State &
     --#         LogBuffer.State from *;
     --# declare suspends => LogBuffer.State;
   is
      pragma Priority (System.Priority'First + 1);
   end LoggerTask;

end Logger;
----------------------------------------------------------------------------
```

```
with LogBuffer;
with SystemLog;

package body Logger
is

   TheLogger : LoggerTask;

   task body LoggerTask
   is
      TheData    : LogBuffer.BufferEntry;
      DataMissed : Boolean;
   begin
      loop
          LogBuffer.Read (TheData, DataMissed);
          SystemLog.Write (TheData, DataMissed);
      end loop;
   end LoggerTask;

end Logger;
```

Document Control and References

Praxis High Integrity Systems Limited, 20 Manvers Street, Bath BA1 1PX, UK.
Copyright © Praxis High Integrity Systems Limited 2006. All rights reserved.

Changes history

Issue 0.1 (13th January 2003): First draft.

Issue 1.0 (28th May 2003): First Definitive Issue.

Issue 1.1 (11th June 2003): Definitive issue following review.

Issue 1.2 (12th June 2003): Minor formatting changes for readability.

Issue 1.3 (12th June 2003): Minor corrections to code to match that on CDROM.

Issue 1.4 (6th January 2005): Update company name and re-issue.

Issue 1.5 (22nd November 2005): Line Manager change.

Changes forecast

None

Document references

1 Real-Time Systems and Programming Languages, Alan Burns and Andy Wellings, Third Edition.

2 INFORMED Design Method for SPARK, Praxis High Integrity Systems

3 The SPARK Ravenscar Profile, Praxis High Integrity Systems