High Integrity Ravenscar using SPARK

RavenSPARK

The language for high integrity concurrent programs

Presented by :

Praxis High Integrity Systems

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

What is RavenSPARK?

Addition of concurrency to sequential SPARK

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

Copyright © Praxis High Integrity Systems 2004

Introduction to Ravenscar Profile

Restricted Ada95 tasking model

- Does not address sequential constructs

Goals

- Eliminate non-deterministic constructs
- Support basic real-time building blocks
- Support Static Timing Analysis
- Support Concurrency Model Checking
- Support a simple Ada runtime system that may be formally verified / certified

Copyright © Praxis High Integrity Systems 2004

Standardization of the Profile

Ravenscar Profile should become part of the Ada05 ISO standard

- WG9-Approved Ada Issues 249 and 305 define the Profile and its Restrictions pragmas

Ravenscar Profile has already been implemented by a number of Ada vendors, some to DO-178B level A, eg

- AONIX ObjectAda/Raven™
- Green Hills Software GSTART™
- GNAT-Pro High Integrity Edition™ on VxWorks/Cert™
- University of Madrid Open Ravenscar Kernel

Copyright © Praxis High Integrity Systems 2004

Outline of the Profile

Basic building blocks:

- Fixed set of non-terminating global tasks
- Fixed set of global shared variables
  - Entry-less protected objects
  - Atomic objects
- Fixed set of global synchronization objects
  - Single-entry protected objects
  - Suspension objects
- Fixed set of global interrupt handlers
- Deterministic and high-precision timing facilities
- Deterministic scheduling and locking policies

Copyright © Praxis High Integrity Systems 2004

Real-Time Program Model

Copyright © Praxis High Integrity Systems 2004

Real-time Templates – Periodic Task

- Periodic tasks released using “delay until”
- Accurate timing via package Ada.Real_Time

```
task body T is
 Release_Time: Ada.Real_Time.Time:= Epoch.T_Start;
 Period : Ada.Real_Time.Time_Span :=
                              Ada.Real_Time.Milliseconds(50);
begin
   <initialisation code>
    loop
       delay until Release_Time; -- Deterministic release
       Do_Periodic_Work;
       Release_Time := Release_Time + Period;
    end loop;
end T;
```

Copyright © Praxis High Integrity Systems 2004

Real-time Templates – Aperiodic Task

- No data transfer => Suspension Object

```
package STC renames Ada.Synchronous_Task_Control;
package Globals is
   task T;
   T_SO : STC.Suspension_Object;
…
   task body T is
   begin
     loop
        STC.Suspend_Until_True (Globals.T_SO);
        Do_Aperiodic_Work;
     end loop;
   end T;
…
STC.Set_True (Globals.T_SO); -- Release T now
```

Copyright © Praxis High Integrity Systems 2004

Real-time Templates – Aperiodic Task

- With data transfer => Protected Entry

```
package Globals is
   protected PO is
        procedure Signal (D: in Data);
        entry Wait       (D: out Data);
…
task body T is
   My_Data : Globals.Data;
begin
   loop
      Globals.PO.Wait (My_Data);
      Operate_On (My_Data);
   end loop;
end T;
…
Globals.PO.Signal (The_Data); -- Wake T and pass the data
```

Copyright © Praxis High Integrity Systems 2004

Real-time Templates – Interrupts

- Interrupt handlers are protected procedures and may release tasks

```
protected PO is
   procedure Signal;
   pragma Attach_Handler (Signal, Globals.Port_ID);
   entry Wait (D: out Data);
private
   The_Data : Data := Null_Data;
   Barrier : Boolean := False;
…
   procedure Signal is
   begin
      The_Data := Read_From_Port;
      Barrier := True; -- Wake waiting task and pass the data
      STC.Set_True (T_SO); -- Wake another task too (no data)
   end Signal;
```

Copyright © Praxis High Integrity Systems 2004

Real-time Templates – State Data

State data may be **shared** or **task-private**

- Shared data **should** be **protected** from concurrent access via :
  - Encapsulation in a **protected object**
  - Atomic access via **pragma Atomic**
  - Atomic access as a **suspension object**
- Task-private data need not be protected

Copyright © Praxis High Integrity Systems 2004

Real-time Templates – Shared Data

```
protected Database is
   procedure Put (Name, Address: String; … );
   procedure Get_Latest (D: out Full_Details);
private
   The_Data : array (0..99) of Full_Details;
   Next_Free : Integer := 0;
…
```

Task 1

```
Database.Put
  (Name => “John Smith”,
   Address => “…”
   <other fields> );
```

Task 2

```
Database.Get_Latest
  (Person);
```

mutual exclusion guaranteed

Real-time Scheduling

For static timing analysis, need

- Deterministic **scheduling** policy
  - FIFO_Within_Priorities is required
- Deterministic **timing** policy
  - Absolute & precise wakeup time is required
- Deterministic **locking** policy
  - Priority Ceiling Locking is required

Priority Ceiling locking provides

- Deterministic worst case **blocking** time
- A **minimal** value for this blocking time
- Guaranteed absence of **deadlock**

Copyright © Praxis High Integrity Systems 2004

Ravenscar Profile Summary

A **framework** for analysable real-time systems

- Support for **schedulability analysis**
- Support for **model checking**

Building blocks to **populate** this framework

- Support for **periodic** tasks
- Support for **aperiodic** tasks
  - synchronisation by **software**, or
  - synchronisation by **interrupt**
- Support for **shared data** communication

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

Copyright © Praxis High Integrity Systems 2004

Introduction to SPARK

- Objectives and aims of SPARK
- How SPARK is constructed
- Introduction to annotations

Copyright © Praxis High Integrity Systems 2004

SPARK - Major Design Considerations

- Logical soundness
- Simplicity of formal language definition
- Expressive power
- Security
- Verifiability
- Bounded space and time requirements
- Correspondence with Ada
- Verifiability of compiled code
- Minimal run-time system requirements

Copyright © Praxis High Integrity Systems 2004

Example of an Ambiguity

```
z := F(x) + G(y);
```

Suppose function **F** modifies **y** as a side-effect of its operation. In this case the meaning of the expression depends on whether **F** or **G** is evaluated first.

A rule stating "functions are not permitted to have side effects" simply turns the ambiguity into an insecurity

Copyright © Praxis High Integrity Systems 2004

Constructing a Safe Subset

- Selection of base language
- Removal of the most deficient language features
- Limitations on the way remaining features may be used
- Introduction of annotations to provide extra information

Copyright © Praxis High Integrity Systems 2004

Constructing a Safe Subset

- Selection of base language
  - Ada (naturally!)
- Removal of the most deficient language features
  - unrestricted tasking
- Limitations on the way remaining features may be used
  - placement of exit and returns stmts
- Introduction of annotations to provide extra information

Copyright © Praxis High Integrity Systems 2004

Why Annotations?

- Annotations strengthen specifications
  - Ada separation of specifications from implementations too weak
- Allows analysis without access to bodies
  - which can be done early on during development
  - even before programs are complete or compilable
- Allows efficient detection of erroneous constructs

Copyright © Praxis High Integrity Systems 2004

An example

```
procedure Inc (X : in out Integer);
--# global in out Callcount;
```

detection of function side-effect

```
function AddOne (X : Integer)
   return Integer is
  XLocal : Integer := X;
begin
  Inc (Xlocal);
  return XLocal;
end AddOne;
```

detection of aliasing

```
Inc (CallCount);
```

Copyright © Praxis High Integrity Systems 2004

Core SPARK annotations

- Subprogram
  - global
  - derives
  - (proof contexts)
- Package
  - own
  - initializes

Copyright © Praxis High Integrity Systems 2004

Package annotations - own, initializes

- indicates that a package contains **state**
- gives that state a **name** that can be used in other annotations
- does **not** reveal implementation details
  - done using **refinement** in package body
- may have mode **in** or **out** indicating connection to the **external environment**
- **initializes** indicates the state is initialized at package **elaboration**

Copyright © Praxis High Integrity Systems 2004

Examples

```
package Stack
--# own State;
--# initializes State;
is
    ...


package Temperature
--# own in Inputs;
is
    function Read return Integer;
    --# global Inputs;
...
```

Copyright © Praxis High Integrity Systems 2004

Subprogram annotations - global

- identifies all data **not locally declared** within the subprogram and **not a parameter** of the subprogram
- transitive
- may have **mode** indicating direction of data flow
- e.g.

```
function IsEmpty return Boolean;
--# global Stack.State;
```

Copyright © Praxis High Integrity Systems 2004

Subprogram annotations - derives

- used on **procedures only**
- amplifies global annotation by specifying **information flow dependencies**
- may be omitted if information-flow analysis not required
- e.g.

```
procedure Pop (X : out Integer);
--# global in out Stack.State;
--# derives Stack.State,
--#             X from Stack.State;
```

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

Potential Errors using Ravenscar

The Ravenscar Profile is excellent for building deterministic real-time systems BUT…

- **Run-time exceptions** may occur
- **Erroneous** behaviour may occur
- **Implementation-defined** behaviour may occur

These must be **eliminated** in high integrity programs

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions

Runtime exceptions may relate to :

- Sequential code executed by tasks, POs
- Concurrency-related constructs

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions – Sequential Code

An unhandled exception in sequential code in a

- Non-tasking program => always **detectable**
  - –Immediate program termination
  - –Potential for diagnosis and response / recovery
  - –May be eliminated by existing static analysis
- Tasking program => may be **silent** e.g.
  - – **abandonment** of executing an interrupt handler
  - – **termination** of single task whilst others continue
  - – **premature exit** from a protected action
    - may result in **inconsistent** protected data
  - – Need ways to eliminate these by static analysis

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions – Concurrency (1)

Two Ada95 concurrency checks apply to Ravenscar programs, and may raise **Program_Error** exception

- Priority Ceiling violation
  - "The active priority of the caller of a protected action must not exceed the callee's ceiling priority"
  - This implements the deterministic **Ceiling_Locking** policy
- Suspension Object queuing violation
  - "A call to Ada.Synchronous_Task_Control.Suspend_Until_True for a suspension object is not allowed if there is already a task suspended on that suspension object"
  - This avoids formation of non-deterministic **queues** of tasks

Examples follow…

Runtime Exceptions – Ceiling Check

```
protected PO is
   pragma Priority(20);
   procedure Signal (D: in Data);
   entry Wait        (D: out Data);
…
task type T (P: System.Priority) is
   pragma Priority (P);
end T;
task body T is
begin
   loop
      Globals.PO.Wait (My_Data);
      Operate_On (My_Data);
   end loop;
end T;
T_OK   : T(20); -- Priority not greater than PO's ceiling
T_Fail : T(21); -- Raises Program_Error on call to PO.Wait
```

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions – SO queue check

```
package Globals is
   SO_Shared : STC.Suspension_Object;
   …
task type T is …
end T;
task body T is
begin
   loop
      STC.Suspend_Until_True(Globals.SO_Shared);
      Response_Actions;
   end loop;
end T;
T_1, T_2: T;
-- Program_Error if either task calls Suspend_Until_True when the
-- other is already suspended on the same call.
```

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions – Concurrency (2)

Two new concurrency checks are introduced to Ravenscar programs, and may raise **Program_Error**

- Potentially-Blocking operation violation
  - "The execution of a protected action must not invoke a potentially-blocking operation"
  - This is needed to support **timing analysis** using Ceiling_Locking, avoiding suspension whilst holding the lock
- Protected Entry queuing violation
  - "The maximum number of calls that are queued on a protected entry must not exceed one"
  - This avoids formation of non-deterministic **queues** of tasks

Examples follow...

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions – Blocking ops

```
protected PO is
   procedure Signal_Program_Error;
   entry Wait (D: out Data);
private
   Protected_Data : Data := Null_Data;
   Barrier : Boolean := False;
end PO;
…
   procedure Signal_Program_Error is
   begin
      PO2.Wait (Protected_Data);   -- Protected entry call
      Ada.Text_IO.Put (Protected_Data);
                  -- Less obvious potentially-blocking operation
      Barrier := True;
   end Signal_Program_Error;
```

Copyright © Praxis High Integrity Systems 2004

Runtime Exceptions – PO queue check

```
protected Server is
   entry Supply    (R: out Resource);
   procedure Free (R: in Resource);
…
task body Client_1 is
begin
   loop
      Server.Supply (My_Resource);
      …
      Server.Free (My_Resource);
   end loop;
end Client_1;
task body Client_2 <also calls Supply and Free>
```

-- Program_Error is raised by the call to Server.Supply
-- if the other client task is also suspended on this call.
-- (This is allowed in full Ada95).

Copyright © Praxis High Integrity Systems 2004

Erroneousness – Elaboration Order

- Global data may be initialised by library package **body elaboration code**
- Correct elaboration **order** is essential if one variable's initialisation depends on another's
  - This dependency is not allowed in seq'tial SPARK
- In full Ada95, order is controlled by **pragmas**
  - This is also true for Ravenscar programs ...
  - ... but you still have to get the order right ...
  - ... and **concurrency** makes things much harder!

Copyright © Praxis High Integrity Systems 2004

Elaboration Order – Non-Tasking

```
package Global_1 is
   N : Natural := 0;
   pragma Elaborate_Body;
end Global_1;

package body Global_1 is
begin
   Ada.Text_IO.Put ("How many records?");
   Ada.Text_IO.Get (N); -- Dynamic initialisation (not SPARK)
end Global_1;


with Global_1;
package body Global_2 is
   -- Global_1 body must be elaborated by now for this to work
   Buffer_Size : Natural := Global_1.N * 256;
…
end Global_2;
```

Copyright © Praxis High Integrity Systems 2004

Elaboration Order – Tasking (1)

For tasking programs, library unit elaboration order is complicated due to:

- Tasks being activated **during** elaboration code
- Tasks continuing to be eligible to execute after their activation is complete (**still during** elaboration code)
- Interrupts potentially being enabled as soon as their handlers are attached (**again during** elaboration code)

Copyright © Praxis High Integrity Systems 2004

Elaboration Order – Tasking (2)

```
package Global_1 is
   pragma Elaborate_Body;  -- Safe for sequential code
   N : Natural;
   task T;
end Global_1;

package body Global_1 is
   task body T is
      Buffer_Size : Natural := N * 256;
   begin …
   end T;
begin  -- T's body will execute here, and Buffer_Size will be ??
   Ada.Text_IO.Put ("How many records?");
   Ada.Text_IO.Get (N);
end Global_1;
```

Copyright © Praxis High Integrity Systems 2004

Elaboration Order – Tasking (3)

Typical software execution model assumes :

- All initialisation code will be executed **first**
- All tasks and interrupt handlers will start **after** the initialisation code is complete

BUT in Ada95 and Ravenscar,
can get **race conditions** during elaboration

Races must be prevented for determinism

- Addressed in Ada05 by new config'n pragma
  - **Partition_Elaboration_Policy (Sequential)**
  - BUT this pragma is **not** required by the Profile

Erroneousness – Shared Variables

A **shared variable** is one that is accessible by more than one task

- The program is **erroneous** if a shared variable is not protected from concurrent access
  - A reader task may read a partially-written value
- Use of unprotected shared variables must be eliminated in high integrity programs

Copyright © Praxis High Integrity Systems 2004

Unprotected Shared Variables

```
package Database is -- unprotected
   procedure Put (Name, Address: String; … );
   procedure Get_Latest (D: out Full_Details);
private
   The_Data : array (0..99) of Full_Details;
   Next_Free : Integer := 0;
…
```

Task 1

```
Database.Put
   (Name => “John Smith”,
    Address => “…”
    <other fields> );
```

Task 2

```
Database.Get_Latest
   (Person);
```

may receive “John Smith” at the wrong address

Copyright © Praxis High Integrity Systems 2004

Incomplete Programs

Ada83 had a problem with
"**lost**" package bodies

- Possible to build legal program that accidentally **omits** a package body
  - Needed elaboration code is not performed
  - Solved in Ada95 by new rules

Ada95 / Ravenscar has a similar problem with "**lost**" packages

Copyright © Praxis High Integrity Systems 2004

Lost Packages (1)

```
package Task_and_Int_Handler is
   task T_1;
   protected P_1 is
      procedure Handler;
      pragma Attach_Handler (Handler, Device);
   end P_1;
end Task_and_Int_Handler;

<lots of other packages>

with <lots of other packages>;
   -- Oops, we forgot to "with" Task_and_Int_Handler here
procedure Main is
   …
end Main;
```

Copyright © Praxis High Integrity Systems 2004

Lost Packages (2)

This is more likely to occur in Ravenscar :

- For a package that only declares **tasks** and / or **interrupt handlers**
  - These must be in **library-level packages**
  - A task has no **callable** entries
  - An interrupt handler is not (usually) **called** by the software
  - Thus there may be no **reference** from main's closure to cause an illegal build
  - Program executes with **missing** components

Copyright © Praxis High Integrity Systems 2004

Implementation-Defined Behaviour

Ravenscar includes No_Task_Termination

- Tasks are assumed to be **“run-forever”**
- Violation of this restriction causes **implementation-defined** response
- Ada Issue 366 has attempted to **standardise** on centralised task termination handling
  - Two different proposals but no consensus
  - Cannot rely on any standard termination response model

Copyright © Praxis High Integrity Systems 2004

Task Termination – Catch-all handler

Existing techniques:

- Catch-all exception handler
  - "when others" handler for each task body
  - "when others" handler for each protected action
- Issues
  - doesn't detect non-exception task termination
  - relies on support for full exception handling features in high integrity environment
  - difficult to map to a centralised health monitor or fault manager model

Copyright © Praxis High Integrity Systems 2004

Task Termination – Controlled Object

- Local Controlled object to do finalization
  - declared in each task body

```
type Termination is new
   Ada.Finalization.Limited_Controlled
      with null record;
procedure Finalize (O: in out Termination) is …;
task body T is
   Final_T : Termination;
begin
   loop
      …
   end loop;
end T;   -- implicit call to Finalize (Final_T);
```

Copyright © Praxis High Integrity Systems 2004

Task Termination – Controlled Object

- Issues
  - relies on support for object-oriented features in high integrity, including implicit dynamic dispatch
  - relies on support for nested finalisation i.e. absence of No_Nested_Finalization restriction
  - at best described as an "error-prone hack"
    - not suitable for reliable software

Best to be able to prove
that tasks do not terminate

Copyright © Praxis High Integrity Systems 2004

Potential Errors - Summary

In order to use the Ravenscar Profile in high integrity, high reliability, safety critical systems, must statically eliminate :

- Potential **exceptions**
  - in sequential and concurrent constructs
- Potential **erroneous execution**
- Uncertainties due to **implementation-defined** behaviour

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

Copyright © Praxis High Integrity Systems 2004

Agenda

1. What is RavenSPARK
2. Introduction to Ravenscar Profile
3. Introduction to SPARK
4. Potential errors in using Ravenscar Profile

break

5. RavenSPARK in practice

Copyright © Praxis High Integrity Systems 2004

RavenSPARK in practice

- Program building blocks
  - tasks
  - protected objects
  - interrupt handlers
- Putting the pieces together
  - sample program
  - error detection
- Partition-wide flow analysis
  - sample design

Copyright © Praxis High Integrity Systems 2004

Building blocks

- Tasks
  - periodic
  - aperiodic
- Protected state
  - atomic variables
  - suspension objects
  - protected objects
- Interrupt handlers

Copyright © Praxis High Integrity Systems 2004

Building blocks

- Tasks
  - periodic
  - aperiodic
- Protected state
  - atomic variables
  - suspension objects
  - protected objects
- Interrupt handlers

Copyright © Praxis High Integrity Systems 2004

Task declaration

- Task object is an instance of a named task type
- It must be declared in a library-level package
  - therefore it is an own variable and
  - must appear in the package's own variable annotation with keyword **task**
- The task type must:
  - be declared in the package specification
  - have a global/derives annotation
  - have an explicit and static priority

Copyright © Praxis High Integrity Systems 2004

Task declaration

```
package P
--# own task T : TT;
is

   task type TT
   --# global in out Q.State;
   --# derives Q.State from Q.State;

   is
      pragma Priority (10);
   end TT;
end P;

package body P
is

   T : TT;

   task body TT is separate;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Task declaration

```
package P
--# own task T : TT;
is
private
   task type TT
   --# global in out Q.State;
   --# derives Q.State from Q.State;

   is
      pragma Priority (10);
   end TT;
end P;

package body P
is

   T : TT;

   task body TT is separate;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Task declaration

```
package P
--# own task T : TT;
is
private
   task type TT
   --# global in out Q.State;
   --# derives Q.State from Q.State;

   is
      pragma Priority (Data.TT_Priority);
   end TT;
end P;

package body P
is

   T : TT;

   task body TT is separate;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Task declaration

```
package P
--# own task T : TT;
is
private
   task type TT (Pr : Integer)
   --# global in out Q.State;
   --# derives Q.State from Q.State;

   is
      pragma Priority (Pr);
   end TT;
end P;

package body P
is
   subtype ST is TT (10);
   T : ST;

   task body TT is separate;
end P;
```

Actual discriminant value must be static

Task declaration (aperiodic)

```
package P
--# own task T : TT;
is
private
   task type TT (Pr : Integer)
   --# global in out Q.State;
   --# derives Q.State from Q.State;
   --# declare (suspends => Q.Event);
   is
      pragma Priority (Pr);
   end TT;
end P;

package body P
is
   subtype ST is TT (10);
   T : ST;

   task body TT is separate;
end P;
```

Suspends is the only property a task may have. It is mandatory if the task suspends by calling a protected entry or by calling Suspend_Until_True.

Copyright © Praxis High Integrity Systems 2004

Avoiding race conditions at start up

- A task (other than the environment task) may not import unprotected state variables; and
- unprotected variables exported by the task may not appear in their package's initializes annotation.

These rules mean:

- A task is solely responsible for initializing any unprotected state that it uses (and which, by definition, it is the sole user of).

Copyright © Praxis High Integrity Systems 2004

Task bodies

- A task body may need a second, refined, annotation exactly like a subprogram
- Task bodies are unchanged from Ada except:
  - they must end with a plain loop (i.e. loop without an iteration scheme or exit statement)
  - the above rule only applies to the main program (the environment task) if there are no other tasks in the program
  - this is part of enforcing the No_Task_Termination restriction

Copyright © Praxis High Integrity Systems 2004

Building blocks

- Tasks
  - periodic
  - aperiodic
- Protected state
  - atomic variables
  - suspension objects
  - protected objects
- Interrupt handlers

Copyright © Praxis High Integrity Systems 2004

Protected state

- Provides the means by which program threads may communicate
- May only be declared at library level
- Is indicated by the keyword **protected** in own variable declarations
  - may also have associated **property** list
  - must be initialized at declaration
- May be implemented as:
  - objects of a pragma Atomic type
  - objects of the predefined suspension object
  - objects of a (user-defined) protected type

Copyright © Praxis High Integrity Systems 2004

Protected declaration - pragma Atomic

```
package P
--# own protected X;
is
    --  operations on X

end P;

package body P
is
    type T is range 1 .. 10;
    pragma Atomic (T);
    X : T := 1;

end P;
```

X *may* have the protected qualifier because it is of Atomic type T

Protected state must be initialized at declaration

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Suspension Obj.

```
package P
--# own protected X (suspendable);
is
    --  operations on X

end P;

package body P
is

    X : Ada.Synchronous_Task_Control.Suspension_Object;

end P;
```

A task can suspend on an object of the predefined suspension object type

Suspension objects are implicitly initialized to False by the Ada run- time

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Protected Object

```
package P
--# own protected X : PT (priority => 10);
is
   protected type PT is
      pragma Priority (10);
      procedure Modify;
      --# global in out PT;
      --# derives PT from PT;
   private
      State : Integer := 0;
   end PT;
end P;

package body P
is


   X : PT;
   protected body PT is separate;
end P;
```

A priority property is mandatory (this is the **Ceiling** priority of the PO)

Priority pragma is mandatory.

All protected components must be statically initialised.

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Protected Object

```
package P
--# own protected X : PT (priority => 10);
is
   protected type PT is
      pragma Priority (10);
      procedure Modify;
      --# global in out PT;
      --# derives PT from PT;
   private
      State : Integer := 0;
   end PT;
end P;
```

Note special use of the type name here; this acts to identify the particular object of type PT that appears in a statement ...

```
package body P
is



   X : PT;
   protected body PT is separate;
end P;
```

... so a call to X.Modify has the effective annotation:
`--# derives X from X;`

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Protected Object

```
package P
--# own protected X : PT (priority => Q.C);
is
   protected type PT is
      pragma Priority (Data.PT_Pr);
      procedure Modify;
      --# global in out PT;
      --# derives PT from PT;
   private
      State : Integer := 0;
   end PT;
end P;

package body P
is



   X : PT;
   protected body PT is separate;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Protected Object

```
package P
--# own protected X : PT (priority => 10);
is
   protected type PT (Pr : Integer) is
      pragma Priority (Pr);
      procedure Modify;
      --# global in out PT;
      --# derives PT from PT;
   private
      State : Integer := 0;
   end PT;
end P;

package body P
is

   subtype SPT is PT (10);
   X : SPT;
   protected body PT is separate;
end P;
```

Must be same value

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Protected Object

```
package P
--# own protected X : PT (priority => 10, suspendable);
is
   protected type PT (Pr : Integer) is
      pragma Priority (Pr);
      entry Modify;
      --# global in out PT;
      --# derives PT from PT;
   private
      State : Integer := 0;
   end PT;
end P;

package body P
is

   subtype SPT is PT (10);
   X : SPT;
   protected body PT is separate;
end P;
```

The entry makes it possible for a task to suspend on an object of type PT

Copyright © Praxis High Integrity Systems 2004

Virtual protected components

- A protected object may **not** access unprotected state; however,
- sometimes we want a protected object to protect access to externally-imported state (e.g. an I/O port)
- The **protects property** allows a protected type to own and protect such state
- The protected state becomes a **virtual protected component** and is inaccessible other than through its protecting PO

Copyright © Praxis High Integrity Systems 2004

Protected declaration - Protected Object

```
package P
--# own in Port;
--#     protected X : PT (priority => 10,
--#                       protects => Port);
is
   protected type PT is
      pragma Priority (10);
      procedure Read (Val : out Integer);
      --# global in PT;
      --# derives Val from PT;
   end PT;
end P;

package body P
is
   Port : Integer; for Port'Address use ...
   X : PT;
   protected body PT is separate;
end P;
```

We check that Port can only be accessed from PT operations

Copyright © Praxis High Integrity Systems 2004

Building blocks

- Tasks
  - periodic
  - aperiodic
- Protected state
  - atomic variables
  - suspension objects
  - protected objects
- Interrupt handlers

Copyright © Praxis High Integrity Systems 2004

Interrupt handlers

- Interrupt procedures declared inside protected type
- Can’t be called as normal procedures
- Protected type’s property list:
  - indicates that the type contains handlers
  - optionally gives a user-supplied name for the external origin of the interrupt

Interrupt handlers

```
package P
--# own protected Events : PT
--#       (priority => 31,
--#        interrupt);
is
   protected type PT is
      pragma Interrupt_Priority (31);

      procedure Alarm;
      --# global in out Q.State;
      --# derives Q.State from Q.State;
      pragma Attach_Handler (Alarm, ...);
   end PT;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Interrupt handlers

```
package P
--# own protected Events : PT
--#        (priority => 31,
--#         interrupt);
is
   protected type PT (IID : IID_Type) is
      pragma Interrupt_Priority (31);

      procedure Alarm;
      --# global in out Q.State;
      --# derives Q.State from Q.State;
      pragma Attach_Handler (Alarm, IID);
   end PT;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Interrupt handlers

```
package P
--# own protected Events : PT
--#      (priority => 31,
--#       interrupt => (Alarm => FireAlarmButton));
is
   protected type PT is
      pragma Interrupt_Priority (31);

      procedure Alarm;
      --# global in out Q.State;
      --# derives Q.State from Q.State;
      pragma Attach_Handler (Alarm, ...);
   end PT;
end P;
```

Copyright © Praxis High Integrity Systems 2004

Protected Bodies

- Operations have second, refined annotation in terms of the **protected elements** of the type

```
procedure Modify;
--# global in out PT;
--# derives PT from PT;
```

```
procedure Modify
--# global in out State;
--# derives State from *;
is
begin
   State := State + 1;
end Modify;
```

The type name is replaced by the actual element name(s)

Copyright © Praxis High Integrity Systems 2004

Additional subprogram annotations

```
--# declare suspends => name
```

must appear on an (unprotected) procedure that calls an **entry** or **Suspend_Until_True**

```
--# declare delay;
```

must appear on an (unprotected) procedure that may execute a **delay** statement or perform an other **blocking** operation (e.g. I/O call)

Copyright © Praxis High Integrity Systems 2004

RavenSPARK in practice

- Program building blocks
  - tasks
  - protected objects
  - interrupt handlers
- Putting the pieces together
  - sample program
  - error detection
- Partition-wide flow analysis

Copyright © Praxis High Integrity Systems 2004

Example program - a stopwatch

Copyright © Praxis High Integrity Systems 2004

Outline design

- A user interface
  - handles button presses
  - starts and stops timer
  - clears display
- A timing loop
  - updates display each second when running
- A display handler
  - maintains the second count
  - causes it to be displayed

Copyright © Praxis High Integrity Systems 2004

Outline design

- A user interface **(protected object)**
  - handles button presses
  - starts and stops timer
  - clears display
- A timing loop **(task)**
  - updates display each second when running
- A display handler **(protected object)**
  - maintains the second count
  - causes it to be displayed

Copyright © Praxis High Integrity Systems 2004

User interface

```
with TuningData;
--# inherit Timer, Display, TuningData;
package User
--# own protected Buttons : PT (Interrupt =>
--#                          (StartClock => StartButton,
--#                           StopClock  => StopButton,
--#                           ResetClock => ResetButton),
--#             Priority => TuningData.UserPriority);
is
   protected type PT is
      pragma Interrupt_Priority (TuningData.UserPriority);

      procedure StartClock;
      --# global in out Timer.Operate;
      --# derives Timer.Operate from *;
      pragma Attach_Handler (StartClock, 1);

      procedure StopClock ...;

      procedure ResetClock ...;
      --# global in out Display.State;
      --# derives Display.State from *;
      pragma Attach_Handler (ResetClock, 3);
   end PT;
end User;
```

Copyright © Praxis High Integrity Systems 2004

User interface - Interrupts

```
with TuningData;
--# inherit Timer, Display, TuningData;
package User
--# own protected Buttons : PT (Interrupt =>
--#                          (StartClock => StartButton,
--#                           StopClock  => StopButton,
--#                           ResetClock => ResetButton),
--#         Priority => TuningData.UserPriority);
is
   protected type PT is
      pragma Interrupt_Priority (TuningData.UserPriority);

      procedure StartClock;
      --# global in out Timer.Operate;
      --# derives Timer.Operate from *;
      pragma Attach_Handler (StartClock, 1);

      procedure StopClock ...;

      procedure ResetClock ...;
      --# global in out Display.State;
      --# derives Display.State from *;
      pragma Attach_Handler (ResetClock, 3);
   end PT;
end User;
```

User interface – Tuning Data

```
with TuningData;
--# inherit Timer, Display, TuningData;
package User
--# own protected Buttons : PT (Interrupt =>
--#                      (StartClock => StartButton,
--#                       StopClock  => StopButton,
--#                       ResetClock => ResetButton),
--#          Priority => TuningData.UserPriority);
is
   protected type PT is
      pragma Interrupt_Priority (TuningData.UserPriority);

      procedure StartClock;
      --# global in out Timer.Operate;
      --# derives Timer.Operate from *;
      pragma Attach_Handler (StartClock, 1);

      procedure StopClock ...;

      procedure ResetClock ...;
      --# global in out Display.State;
      --# derives Display.State from *;
      pragma Attach_Handler (ResetClock, 3);
   end PT;
end User;
```

Copyright © Praxis High Integrity Systems 2004

User interface - Interfaces

```
with TuningData;
--# inherit Timer, Display, TuningData;
package User
--# own protected Buttons : PT (Interrupt =>
--#                     (StartClock => StartButton,
--#                      StopClock  => StopButton,
--#                      ResetClock => ResetButton),
--#         Priority => TuningData.UserPriority);
is
   protected type PT is
      pragma Interrupt_Priority (TuningData.UserPriority);

      procedure StartClock;
      --# global in out Timer.Operate;
      --# derives Timer.Operate from *;
      pragma Attach_Handler (StartClock, 1);

      procedure StopClock ...;

      procedure ResetClock ...;
      --# global in out Display.State;
      --# derives Display.State from *;
      pragma Attach_Handler (ResetClock, 3);
   end PT;
end User;
```

Copyright © Praxis High Integrity Systems 2004

Timer (1)

```
with TuningData;
--# inherit Ada.Synchronous_Task_Control,
--#         Ada.Real_Time, Display, TuningData;
package Timer
--# own protected Operate (suspendable);
--#     task TimingLoop : TT;
is
   procedure StartClock;
   --# global in out Operate;
   --# derives Operate from *;

   procedure StopClock;
   --# global in out Operate;
   --# derives Operate from *;

...
```

Copyright © Praxis High Integrity Systems 2004

Timer (1) – Internal Protected Variable

```
with TuningData;
--# inherit Ada.Synchronous_Task_Control,
--#         Ada.Real_Time, Display, TuningData;
package Timer
--# own protected Operate (suspendable);
--#     task TimingLoop : TT;
is
   procedure StartClock;
   --# global in out Operate;
   --# derives Operate from *;

   procedure StopClock;
   --# global in out Operate;
   --# derives Operate from *;

...
```

Copyright © Praxis High Integrity Systems 2004

Timer (2) – Internal Task

```
package Timer
--# own protected Operate (suspendable);
--#     task TimingLoop : TT;
is
...

private
   task type TT
   --# global in out Operate,
   --#               Display.State;
   --#        in     Ada.Real_Time.ClockTime;
   --# derives Operate,
   --#         Display.State from * &
   --#         null          from
   --#           Ada.Real_Time.ClockTime;
   --# declare suspends => Operate;
   is
      pragma Priority (TuningData.TimerPriority);
   end TT;
end Timer;
```

Copyright © Praxis High Integrity Systems 2004

Timer (2) – Internal Synchronisation

```
package Timer
--# own protected Operate (suspendable);
--#     task TimingLoop : TT;
is
...

private
   task type TT
   --# global in out Operate,
   --#               Display.State;
   --#        in     Ada.Real_Time.ClockTime;
   --# derives Operate,
   --#         Display.State from * &
   --#         null          from
   --#           Ada.Real_Time.ClockTime;
   --# declare suspends => Operate;
   is
      pragma Priority (TuningData.TimerPriority);
   end TT;
end Timer;
```

Copyright © Praxis High Integrity Systems 2004

Timer body

```
package Timer
--# own protected Operate (suspendable);
--#     task TimingLoop : TT;
is
```

```
with Ada.Synchronous_Task_Control,
  Ada.Real_Time,
  Display;
use type Ada.Real_Time.Time;
package body Timer
is
  Operate : Ada.Synchronous_Task_Control.Suspension_Object;
  TimingLoop : TT;

   procedure StartClock
   is
   begin
      Ada.Synchronous_Task_Control.Set_True (Operate);
   end StartClock;

   procedure StopClock
   is
   begin
      Ada.Synchronous_Task_Control.Set_False (Operate);
   end StopClock;

   ...
```

Copyright © Praxis High Integrity Systems 2004

Timer body

```
   ...

   task body TT is
      Release_Time : Ada.Real_Time.Time;
      Period : constant Ada.Real_Time.Time_Span :=
                                     TuningData.TimerPeriod;
   begin
      Display.Initialize; -- ensure we get 0 on the screen at start up
      loop
         -- wait until user allows clock to run
         Ada. ... .Suspend_Until_True (Operate);
         Ada. ... .Set_True (Operate); -- keep running
         -- once running, count the seconds
         Release_Time := Ada.Real_Time.Clock;
         Release_Time := Release_Time + Period;
         delay until Release_Time;
         -- each time round, update the display
         Display.AddSecond;
      end loop;
   end TT;
end Timer;
```

```
task type TT
 --# declare suspends => Operate;
```

Copyright © Praxis High Integrity Systems 2004

Display – Protected operations on Port

```
with TuningData;
--# inherit TuningData;
package Display
--# own out        Port;
--#     protected State : PT
--#       (priority => TuningData.DisplayPriority,
--#        protects => Port);
is
   procedure Initialize;
   --# global in out State;
   --# derives State from *;

   procedure AddSecond;
   --# global in out State;
   --# derives State from *;

   ...
```

Copyright © Praxis High Integrity Systems 2004

Display (2) – Internal protected type

```
...
private
  protected type PT is
  pragma Interrupt_Priority (TuningData.DisplayPriority);

      -- add 1 second to stored time and send it to port
      procedure Increment;
      --# global in out PT;
      --# derives PT from *;

      -- clear time to 0 and send it to port
      procedure Reset;
      --# global in out PT;
      --# derives PT from *;

   private
      Counter : Natural := 0;
   end PT;
end Display;
```

Copyright © Praxis High Integrity Systems 2004

Display body

```
package Display
--# own out          Port;
--#     protected State : PT
--#        (priority => TuningData. ...,
--#         protects => Port);
```

```
package body Displ
is
   State : PT;
   Port  : Integer; for Port'Address use 16#FFFF_FFFF#;

   protected body PT is
      procedure Increment
      --# global in out Counter; out Port;
      --# derives Port, Counter from Counter;
      is
      begin
         Counter := Counter + 1;
         Port  := Counter;
      end Increment;

      procedure Reset
      --# global out Counter, Port;
      --# derives Counter, Port from ;
      is
      begin
         Counter := 0;
         Port  := Counter;
      end Reset;
   end PT;
```

Port only visible in body of PT

Display body – Visible Operations

```
package body Display
is
   ...

   procedure Initialize
   is
   begin
      State.Reset;            -- call to protected operation
   end Initialize;

   procedure AddSecond
   is
   begin
      State.Increment;        -- call to protected operation
   end AddSecond;

end Display;
```

Copyright © Praxis High Integrity Systems 2004

Error detection

- Annotations plus declaration of types in package specs. allows Ravenscar errors to be detected statically. e.g.
  - we know TimingLoop's priority
  - we know it is of type Timer.TT
  - TT's annotations show that it uses protected object Display.State
  - Display.State's priority is in its property annotation
  - if it is less than TimingLoop's we have a priority ceiling violation

Copyright © Praxis High Integrity Systems 2004

RavenSPARK in practice

- Program building blocks
  - tasks
  - protected objects
  - interrupt handlers
- Putting the pieces together
  - sample program
  - error detection
- Partition-wide flow analysis

Copyright © Praxis High Integrity Systems 2004

Partition-wide flow analysis

- What is partition-wide flow analysis?
  - A definition of the information flow across the entire partition (program)
- A valuable **design aid**
  - Defines the **intended** dependency relations in terms of inputs and outputs
  - Verifies that the **actual** dependency relations correspond to the intended ones

Copyright © Praxis High Integrity Systems 2004

Expected flow analysis

- A new partition annotation is added to the main program definition
  - Similar style to that for procedures and tasks
  - Expected flows described using **global** and **derives** annotations
- Must not be confused with the annotations for the main subprogram itself

Copyright © Praxis High Integrity Systems 2004

Complete stopwatch

```
with User, Timer, Display;
--# inherit User, Timer, Display, Ada.Real_Time;
--# main_program;
--# global in      User.StartButton,
--#                User.StopButton,
--#                User.ResetButton,
--#                Ada.Real_Time.ClockTime;
--#         in out Timer.Operate,
--#                Display.State;
--# derives Timer.Operate from *,
--#                            User.StartButton,
--#                            User.StopButton &
--#          Display.State from *,
--#                            Timer.Operate,
--#                            User.StartButton,
--#                            User.StopButton,
--#                            User.ResetButton &
--#          null          from Ada.Real_Time.ClockTime;
procedure Main
--# derives ;
is pragma Priority (10);
begin
   null;
end Main;
```

Partition annotation

Main procedure annotation

Copyright © Praxis High Integrity Systems 2004

Complete stopwatch-Start/Stop buttons

```
with User, Timer, Display;
--# inherit User, Timer, Display, Ada.Real_Time;
--# main_program;
--# global in      User.StartButton,
--#                User.StopButton,
--#                User.ResetButton,
--#                Ada.Real_Time.ClockTime;
--#         in out Timer.Operate,
--#                Display.State;
--# derives Timer.Operate from *,
--#                             User.StartButton,
--#                             User.StopButton &
--#          Display.State from *,
--#                             Timer.Operate,
--#                             User.StartButton,
--#                             User.StopButton,
--#                             User.ResetButton &
--#          null          from Ada.Real_Time.ClockTime;
procedure Main
--# derives ;
is pragma Priority (10);
begin
   null;
end Main;
```

Copyright © Praxis High Integrity Systems 2004

Complete stopwatch – Reset button

```
with User, Timer, Display;
--# inherit User, Timer, Display, Ada.Real_Time;
--# main_program;
--# global in      User.StartButton,
--#                User.StopButton,
--#                User.ResetButton,
--#                Ada.Real_Time.ClockTime;
--#         in out Timer.Operate,
--#                Display.State;
--# derives Timer.Operate from *,
--#                             User.StartButton,
--#                             User.StopButton &
--#         Display.State from *,
--#                             Timer.Operate,
--#                             User.StartButton,
--#                             User.StopButton,
--#                             User.ResetButton &
--#         null          from Ada.Real_Time.ClockTime;
procedure Main
--# derives ;
is pragma Priority (10);
begin
   null;
end Main;
```

Copyright © Praxis High Integrity Systems 2004

Complete stopwatch – Effect of Timer

```
with User, Timer, Display;
--# inherit User, Timer, Display, Ada.Real_Time;
--# main_program;
--# global in       User.StartButton,
--#                 User.StopButton,
--#                 User.ResetButton,
--#                 Ada.Real_Time.ClockTime;
--#          in out Timer.Operate,
--#                 Display.State;
--# derives Timer.Operate from *,
--#                             User.StartButton,
--#                             User.StopButton &
--#          Display.State from *,
--#                             Timer.Operate,
--#                             User.StartButton,
--#                             User.StopButton,
--#                             User.ResetButton &
--#          null          from Ada.Real_Time.ClockTime;
procedure Main
--# derives ;
is pragma Priority (10);
begin
   null;
end Main;
```

Copyright © Praxis High Integrity Systems 2004

Complete stopwatch (without named interrupts)

```
with User, Timer, Display;
--# inherit User, Timer, Display, Ada.Real_Time;
--# main_program;
--# global in       User.Buttons,
--#                 Ada.Real_Time.ClockTime;
--#         in out Timer.Operate,
--#                 Display.State;
--# derives Timer.Operate,
--#          Display.State from *,
--#                             Timer.Operate,
--#                             User.Buttons &
--#          null          from Ada.Real_Time.ClockTime;
procedure Main
--# derives ;
is pragma Priority (10);
begin
   null;
end Main;
```

Copyright © Praxis High Integrity Systems 2004

Actual flow analysis

- The **global** and **derives** annotations for each "thread" are retrieved
  - A "thread" is :
    - The main procedure
    - Each Ada task
    - Each interrupt handler (protected procedure)
- Then we apply some transformations...

Copyright © Praxis High Integrity Systems 2004

Step one: effect of suspension

- we consider each object a task may suspend on as an import to that task

```
--# derives Operate,
--#         Display.State from * &
--#         null          from
--#           Ada.Real_Time.ClockTime;
--# declare suspends => Operate;
```

becomes

```
--# derives Operate,
--#         Display.State from *, Operate &
--#         null          from Operate,
--#           Ada.Real_Time.ClockTime;
--# declare suspends => Operate;
```

Copyright © Praxis High Integrity Systems 2004

Step two: effect of interrupts

- We consider exports of interrupt handlers to be derived from their PO or from the user-chosen names in the property annotation

```
--# own protected Buttons : PT (Interrupt =>
--#                 (StartClock => StartButton,
...
   procedure StartClock;
   --# global in out Timer.Operate;
   --# derives Timer.Operate from *;
```

becomes

```
   procedure StartClock;
   --# global in out Timer.Operate;
   --# derives Timer.Operate from *, StartButton;
```

Copyright © Praxis High Integrity Systems 2004

Step two: effect of interrupts

- We consider exports of interrupt handlers to be derived from their PO or from the user-chosen names in the property annotation

```
--# own protected Buttons : PT (Interrupt, ...
--#
...

   procedure StartClock;
   --# global in out Timer.Operate;
   --# derives Timer.Operate from *;
```

becomes

```
    procedure StartClock;
    --# global in out Timer.Operate;
    --# derives Timer.Operate from *, Buttons;
```

Step three: combine flow relations

- We combine the flow relations by:
  - taking their union
  - taking the transitive closure of the resulting relation
- These steps capture the effect of one task producing data and another consuming it.

Copyright © Praxis High Integrity Systems 2004

Step four: comparison

- The computed actual flow relation is compared to the intended one and any discrepancies reported
  - This should catch the error caused by “lost” (not with-ed) packages

Copyright © Praxis High Integrity Systems 2004

The Mine Pump Example

- The Mine Pump is a classic example of a real-time system with actuators being set as a result of sensors
- Overview:
  - Pumps water out when level is too high
  - Not operate if the methane level is too high
  - Must support manual override of pump
  - Visual display of state (gas/water/pump)
  - All critical events recorded in a log

Copyright © Praxis High Integrity Systems 2004

Mine Pump System Architecture

AirSensor
Air Monitor
Logger
SystemLog
CarbonMonoxideSensor
Carbon Monoxide Monitor
LogBuffer
MethaneSensor
Methane Monitor
PumpSwitch
PumpSwitchInterrupt
LogBuffer
PumpSwitch
LightPanel
Pump Controller
WaterLevelInterrupt
WaterLevel Sensor
WaterLevelSensor
PBITDone
Pump
LightPanel
WaterFlowSensor
BIT

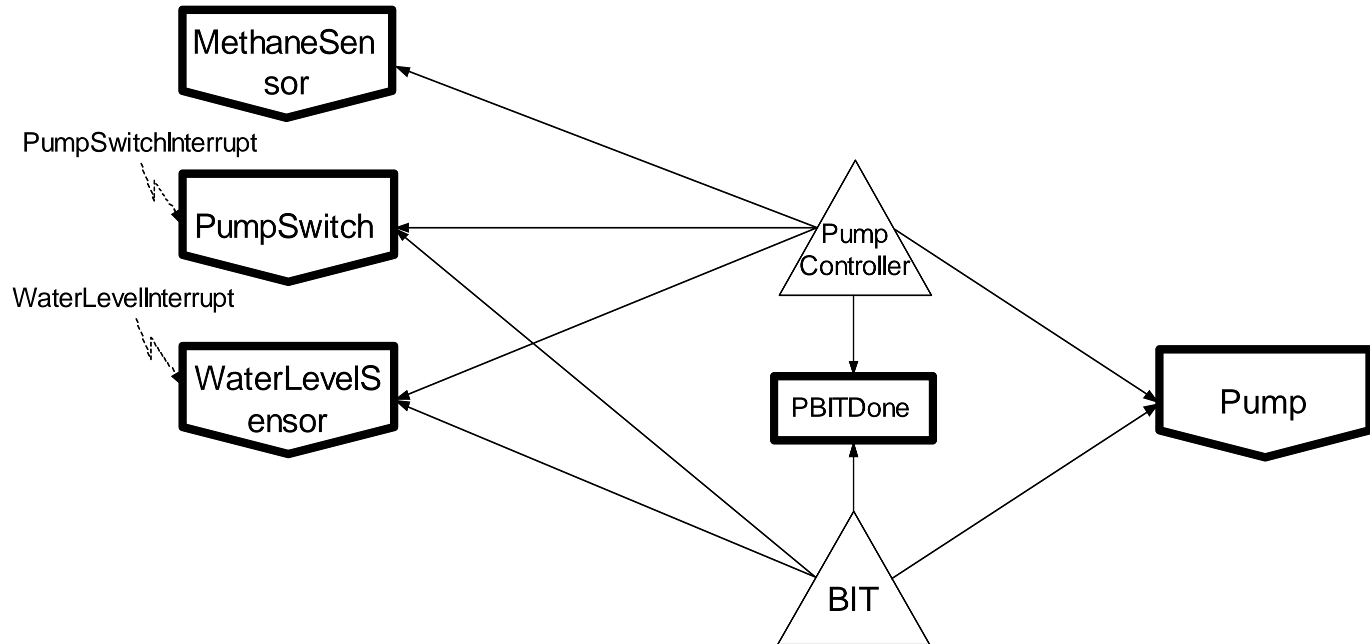
Copyright © Praxis High Integrity Systems 2004

Partition Annotation

- Once the architecture of the system has been defined, the partition annotation can be constructed
  - This represents the **high level design** of the system
  - The actual implementation can be **incrementally checked** against the design via the RavenSPARK annotations

Copyright © Praxis High Integrity Systems 2004

Annotation for Pump




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

Copyright © Praxis High Integrity Systems 2004

Annotation for Light Panel

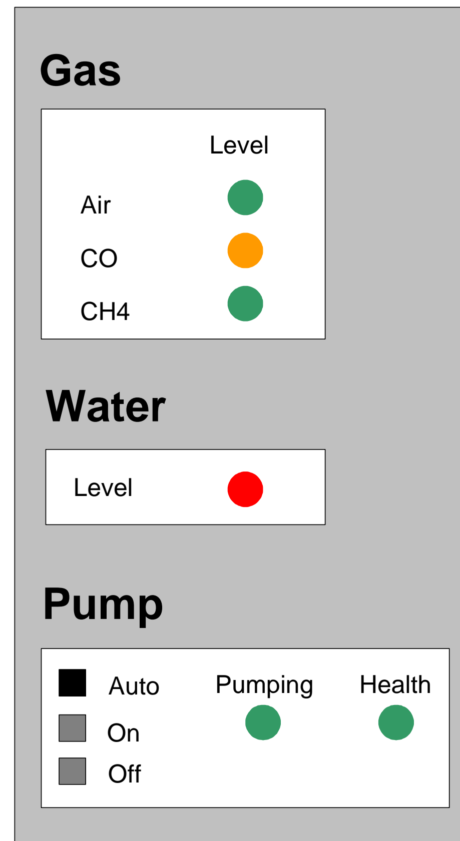
First (crude) iteration might be:

```
--# derives LightPanel.State from
--#             <all sensors>, Pump
```

Copyright © Praxis High Integrity Systems 2004

Annotation for Light Panel (2)

But, a more precise definition could be given if the LightPanel were described in terms of how the sensors affect each individual light:



Copyright © Praxis High Integrity Systems 2004

Final Annotation for Light Panel (3)

This annotation is used to verify that the correct sensor affects each light:

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

Copyright © Praxis High Integrity Systems 2004

Conclusions

- RavenSPARK has preserved the main SPARK principles of:
  - unambiguous definition
  - expressive power of Ada constructs
  - verifiability of design against implem'tation
  - proof of absence of run-time exceptions
  - certifiable run-time system requirements

Copyright © Praxis High Integrity Systems 2004

RavenSPARK

For more information, visit :

www.sparkada.com

or e-mail us at :

sparkinfo@praxis-his.com

Copyright © Praxis High Integrity Systems 2004