SWE30001 Real-Time Programming

2/04/2015

# Lecture 5:

# HRT-HOOD Implementation & Example

Coding HRT-HOOD objects

Example

(Ref: Burns & Wellings 3rd ed pp 658-682)

SWINBURNE UNIVERSITY OF TECHNOLOGY



*Real-Time Programming*

26/03/2007

## Outline

- HRT-HOOD implementation in Ada95
  - General guidelines
  - Cyclic
  - Protected
    - Note on Readers-Writers
  - Sporadic
    - Single item buffer
    - Open issues
- HRT-HOOD case study
  - Problem outline
  - First, second and third level decomposition
  - Example code

2

# HRT-HOOD Implementation

Guidelines:

- Each HRT-HOOD object becomes a package
  - ASM style
- Package specs are minimal
  - Maximise data hiding: details hidden in spec
  - (no private section needed)
- Some packages may be needed for global definitions (spec only)
- Some HRT-HOOD objects may disappear:
  - Composites (A) that simply call-through
- Preserve design in comments

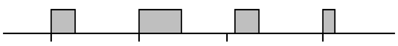


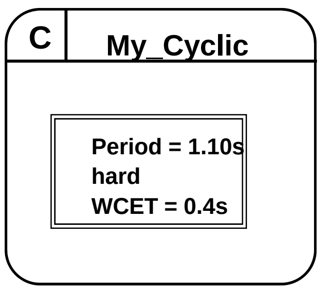




# Cyclic Object - 1

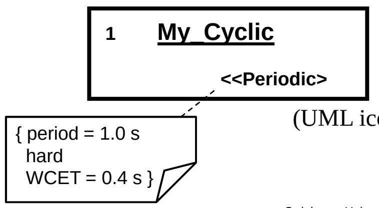
- here simplest form:
  - no operations
  - cannot act as a server or collaborator
- WCET
  - worst case execution time (**cpu** time) of each cycle






(HRT-HOOD icon)

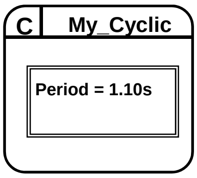
- other real-time attributes:
  - priority
  - deadline (= period by default)



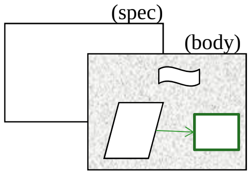

(UML icon)

# Cyclic Implementation

(HRT-HOOD icon)




(Booch icons)




```
package My_Cyclic is  -- CYCLIC
  Period : constant := 1.10;
end My_Cyclic;
```

- No operations defined

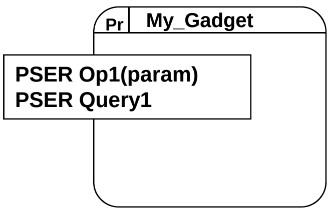
```
with Things;
package body My_Cyclic is  -- CYCLIC
  … -- whatever needed here
  task Thread;
  task body Thread is
      …
  begin
    loop Cyclic_Op;  delay Period;
     end loop;
  end Thread;
end My_Cyclic;
```

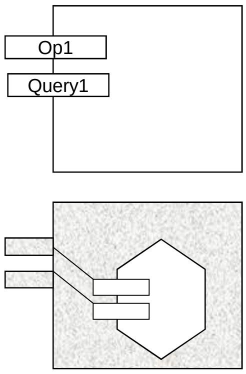






# Protected Implementation -1




- Operations are coded as subprograms that call corresponding ops in an Ada95 protected
- The Ada95 protected is completely hidden in the package body
- Data is inside that

Real-Time Programming
26/03/2007

# Protected Implementation - 2

```
package My_Gadget is
   -- PROTECTED

procedure Op1( -- PSER
   Param1 : in A_Type);

function Query1 -- PSER
  return B_Type;

end My_Gadget;
```

```
-- similar to
procedure Query1(
   B : out B_Type);
```

```
with Various;
package body My_Gadget is – PROTE..
  protected Object is
    procedure Op1(
         Param1 : in A_Type);
    function Query1 return B_Type;
  private
    Data1 : A_Type := initial_value;
    Data2 : …
  end Object;

  procedure Op1(
         Param1 : in A_Type) is
  begin
    Object.Op1(Param1);
  end Op1;
```

7



Real-Time Programming
26/03/2007

# Protected Implementation - 3

```
function Query1 return B_Type is
begin
    return Object.Query1;
end Query1;

protected body Object is
  procedure Op1(
        Param1 : in A_Type) is
  begin
        Data1 := …. ;
        …
  end Op1;

  function Query1 return B_Type is
  begin
    …
    return Data2;
  end Query1;
end Object;

end My_Gadget;
```

procedures allowed to modify data

the real work gets done here!

functions are guaranteed to NOT modify data

Both guaranteed not to block, eg delay

8

Real-Time Programming
26/03/2007

## Note on Readers-Writers Protocol

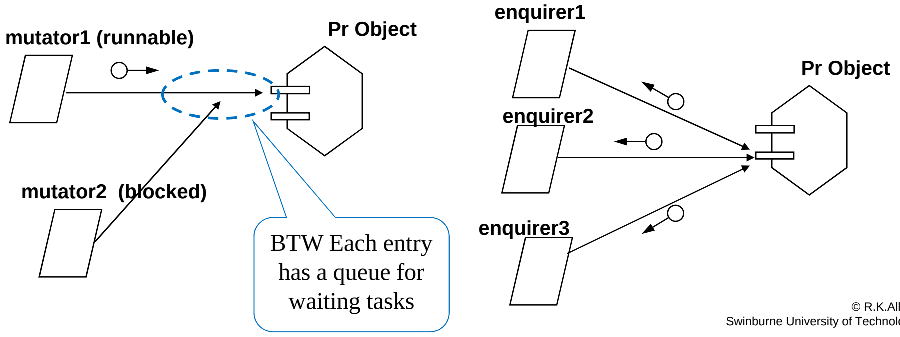
- What threads can be "inside" a protected?

  at any instant

  – none

  – one task that might modify the state -- procedure

  – many tasks (n) that read the state -- function

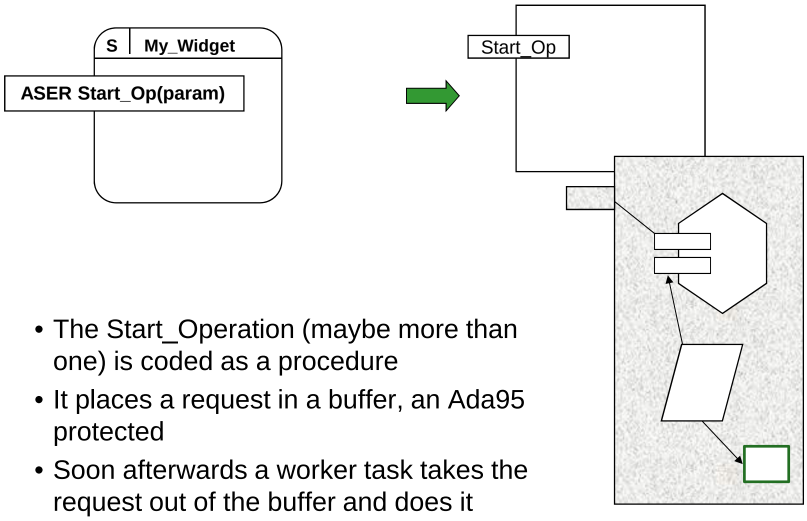



9



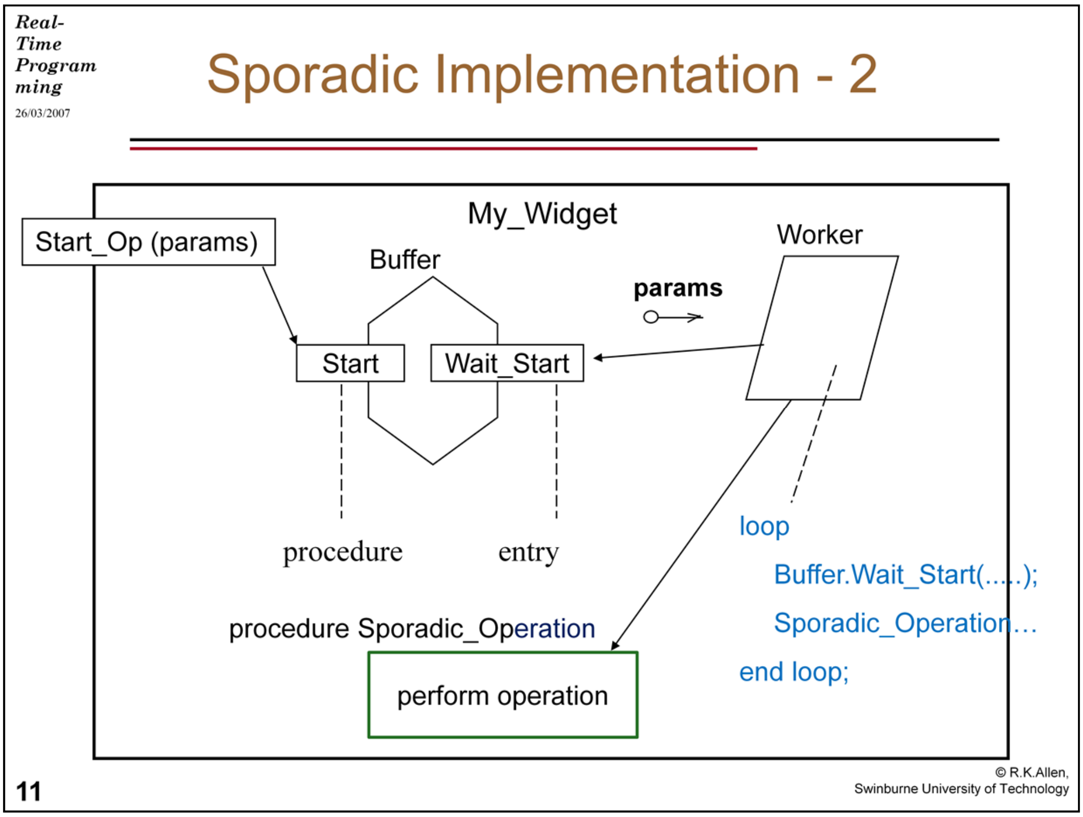
Real-Time Programming
26/03/2007

## Sporadic Implementation - 1




- The Start_Operation (maybe more than one) is coded as a procedure
- It places a request in a buffer, an Ada95 protected
- Soon afterwards a worker task takes the request out of the buffer and does it

10




- How ASER is achieved:
- Buffer.Start is a protected procedure. Its client must not block (except briefly due to mut.ex.).
- The parameters which describe the operation request are stored in the Buffer -- the operation is actually done by the task **Worker**.
- Wait_Start is an entry -- thread Worker blocks until an operation request has arrived.

*Real-Time Programming*

26/03/2007

## Sporadic Implementation - 3

w/o overrun detection

```
-- Single item buffer
protected Buffer is
    procedure Start(
      Request: in Request_Type);

    entry Wait_Start(
      Request: out Request_Type
     );
private
    Item : Request_Type;
    Item_Available
                    : Boolean := False;
end Buffer;
---
protected body Buffer is
    procedure Start(
      Request: in Request_Type) is
    begin
        Item_Available := True;

        Item := Request;
    end Start;

    entry Wait_Start(
        Request: out Request_Type
        )
        when Item_Available is
    begin
        Request := Item;

        Item_Available := False;

    end …
```

12



- Without over run detection.

*Real-Time Programming*
26/03/2007

## Sporadic Implementation - 4

```
-- Single item buffer
protected Buffer is
    procedure Start(
      Request: in Request_Type);

    entry Wait_Start(
      Request: out Request_Type;
      Over_Run : out Boolean );
private
    Item : Request_Type;
    Item_Available,
    Too_Fast       : Boolean := False;
end Buffer;
---
protected body Buffer is
    procedure Start(
      Request: in Request_Type) is
    begin
      if Item_Available then
        Too_Fast := True;
      else
        Item_Available := True;
      end if;
      Item := Request;  -- ignore old
    end Start;

    entry Wait_Start(
      Request: out Request_Type;
      Over_Run : out Boolean )
      when Item_Available is
    begin
      Request := Item;
      Over_Run := Too_Fast;
      Item_Available := False;
      Too_Fast := False;
    end ...
```

13



- Particularly when there is only space to queue up a single request you should write code in case your careful design fails.
  That is, at least detect that something has gone wrong.
- Over_Run is set true when a second call of Start occurs before the previous request has been dealt with.
- Note that if this occurs the older request is lost. To change that behaviour move the line "Item := Request;" into the "else" part above.
- The code that calls Wait_Start is in task **Worker**-- see next slide.

*Real-Time Programming*
19/03/2012

# Sporadic Implementation - 4

```
procedure Sporadic_Operation(
      Request: in Request_Type) is
begin
      ...
end Sporadic_Operation;

task Worker;

task body Worker is
    Req : Request_Type;
     Oops : Boolean := false;
begin
     loop
        Buffer.Wait_Start(Req, Oops);
        if Oops then ...

        Sporadic_Operation(Req);
     end loop;
end Worker;

end Widget; -- package
```

It's good style to separate op code from loop code.

Display error message etc.

**14**

© R.K.Allen, Swinburne University of Technology

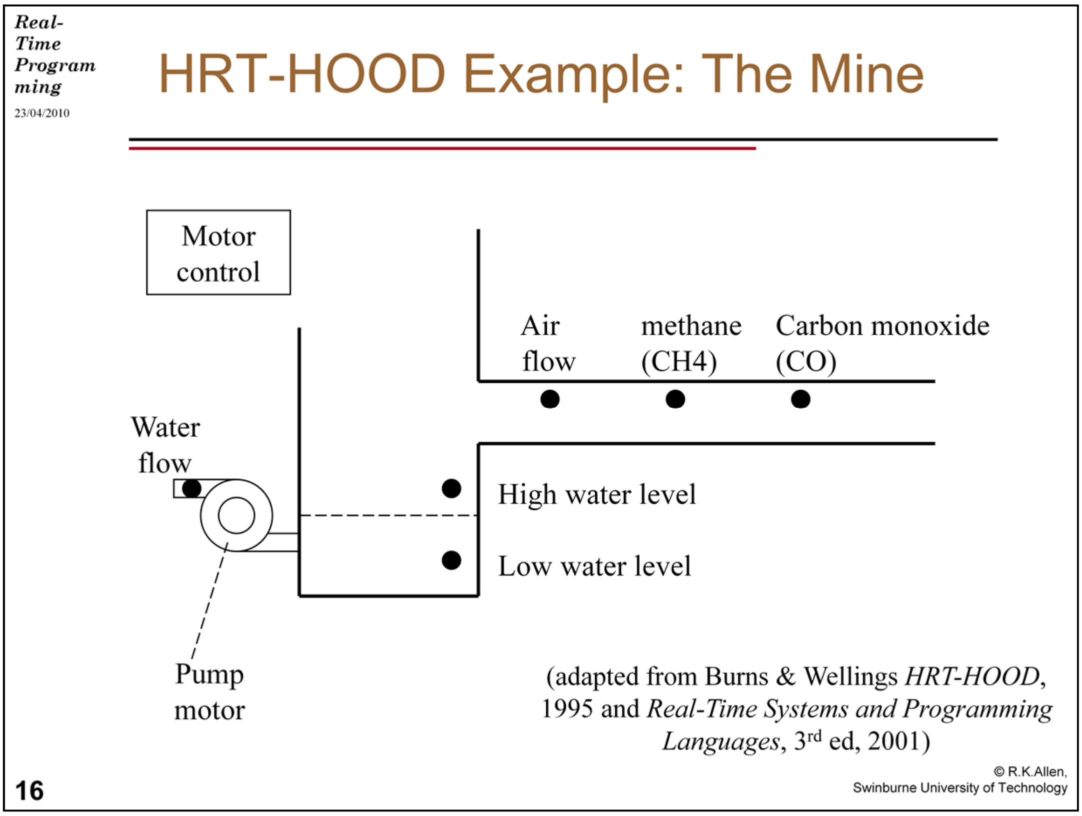
- Worker blocks until a request is available then does it.
- Oops is true on an over-run.
- This version does the operation anyway – it might be the latest or earlier request.
- See last slide this lecture for simple example without over-run handling.

Real-Time Programming
26/03/2007

# Sporadic Implementation - 5

- Notes:
  - The above code does not assume response is complete before the next request arrives – it has single-buffering.
  - It does detect “buffer full” but doesn’t block the client
  - The worker is responsible for error message/cleanup on over-run
- Issues
  - If there is an over-run ignore old data or new data?
  - Longer buffer? How do we implement it?
  - Multiple operations (Op1, Op2, etc)?
  - Where do we put instance variables? None?
  - Multiple identical objects?
- Answers and more details
  - Next week

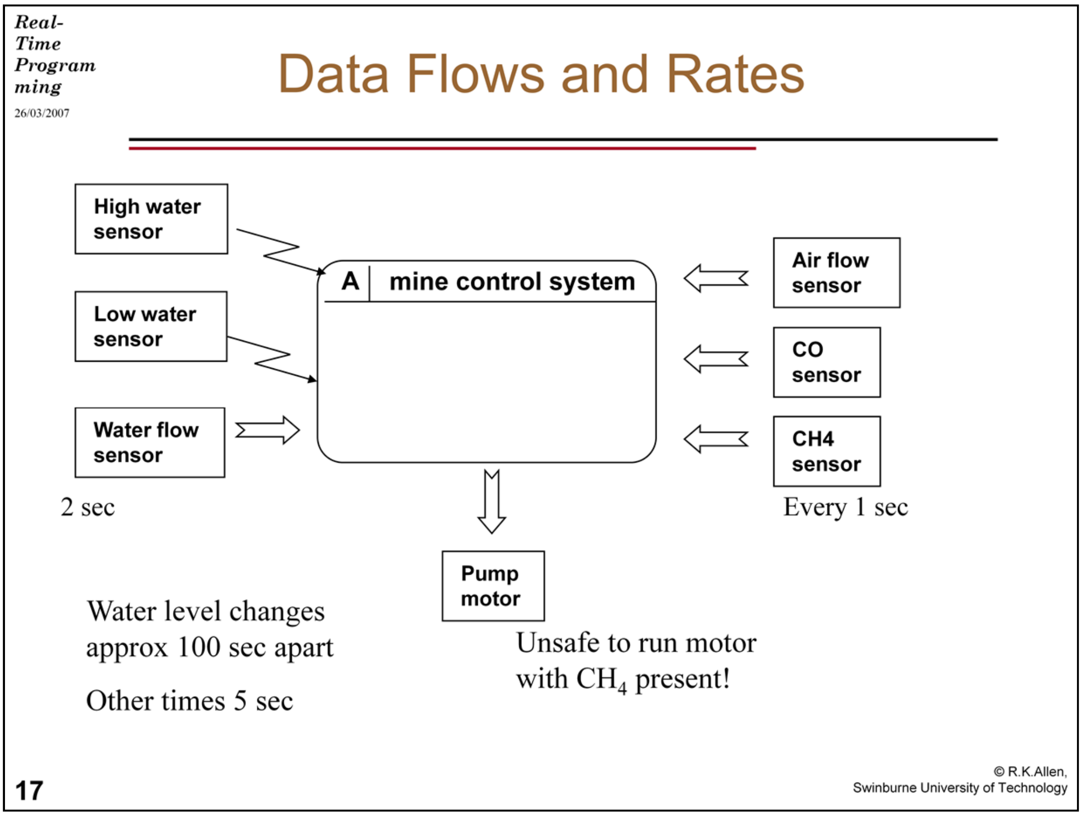
15




- Highly simplified diagram of a coal mine!

  Deep shaft that water seeps into -- must be pumped out to avoid flooding.

  Horizontal tunnel where the coal is extracted and the miners work.
- There is also an air pump, not automatically controlled.




- If methane (chemical formula $CH_4$) level rises then there could be an explosion so it is checked often (every sec) and the deadline is 0.3 second.
- High and Low water sensors produce interrupts no faster than 100 seconds apart (slow because of the large volume of water).
- Water flow sensor confirms that the water pump is working but because it is along the pipe the software will need to wait about 2 seconds after switching on or off the pump motor for any readings to make sense.
- The other sensors can be sampled every 5 sec.
- If CO goes above the safe level then an alarm must sound to evacuate the workers.

[**2012: these times have been changed to be similar to Burns & Wellings 4th ed 2009]**

*Real-Time Programming*
2/04/2015

## Timing Characteristics

| | arrival time | deadline | kind |
|---|---|---|---|
| • CH4 Sensor | 1 sec | 0.3 sec | CYCLIC |
| • CO Sensor | 5 sec | 3 sec | " |
| • Water_Flow | 2 sec | 1 sec | " |
| • Air_Flow | 5 sec | 5 sec | " |

**Poll their devices**

| | | | |
|---|---|---|---|
| • Water level detectors | | | |
| | >= 100 | 20 sec | INTERRUPT |

18



- Complete:

  CH4_Sensor P=1 D=0.3 CYCLIC

  CO Sensor P=5 D=3 CYCLIC

  Water_Flow sensor P=5 D=2 CYCLIC

  Air_Flow_sensor P=5 D=5 CYCLIC

  Water level detectors  Tmin = 100 D=20 interrupts/sporadic

  [**2012: these times have been changed to be similar to  Burns & Wellings 4th ed 2009]**

# Identifying Objects

NEW

- Beyond "underline the noun", look for
  - Physical devices
  - Real world items
  - Causal objects
  - Control elements
  - Service providers
  - Messages and info flow
  - Key concepts
  - Transactions
  - Persistent data
  - Visual elements
- and apply scenarios

(adapted from Douglass, B.P. *Real Time UML* 3rd ed, Addison Wesley 2004)

Real-Time Programming
23/04/2010

## Hierarchical Design Process

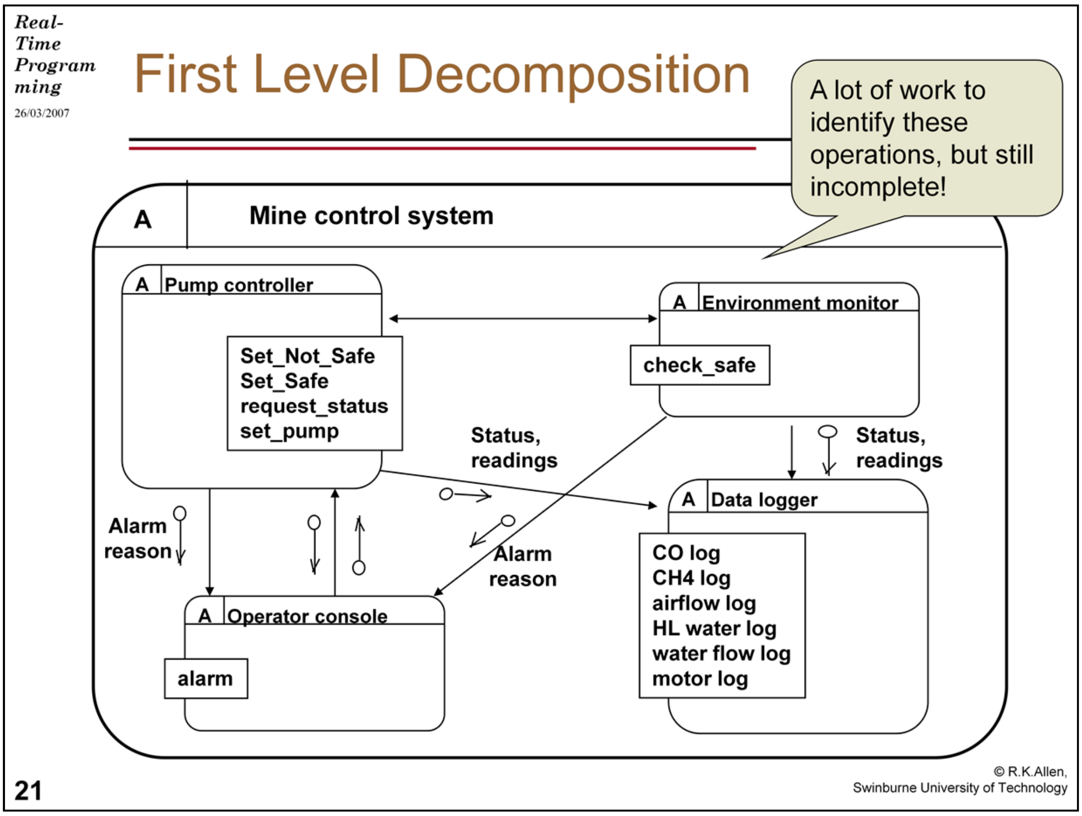
- Starting from the top level object we divide it ("decompose it") into subsystems, objects.
- Each 2nd level object is decomposed…
- etc
- We stop when we have no more composite objects, only "terminal" objects.
- In a fully HRT-HOOD design (here) all terminal objects are non-Active.
  - Cyclic, Passive, Protected, Sporadic
    - (Environment objects are considered to be outside the system being designed.)
- In this unit, we designate/mark composite objects as Active.
- Sometimes it may be necessary to have terminal objects that don't fit the HRT-HOOD types.
  - terminal Active
  - Possibly can't be analyzed, so maybe only safe to use these for "background", low priority tasks.

20

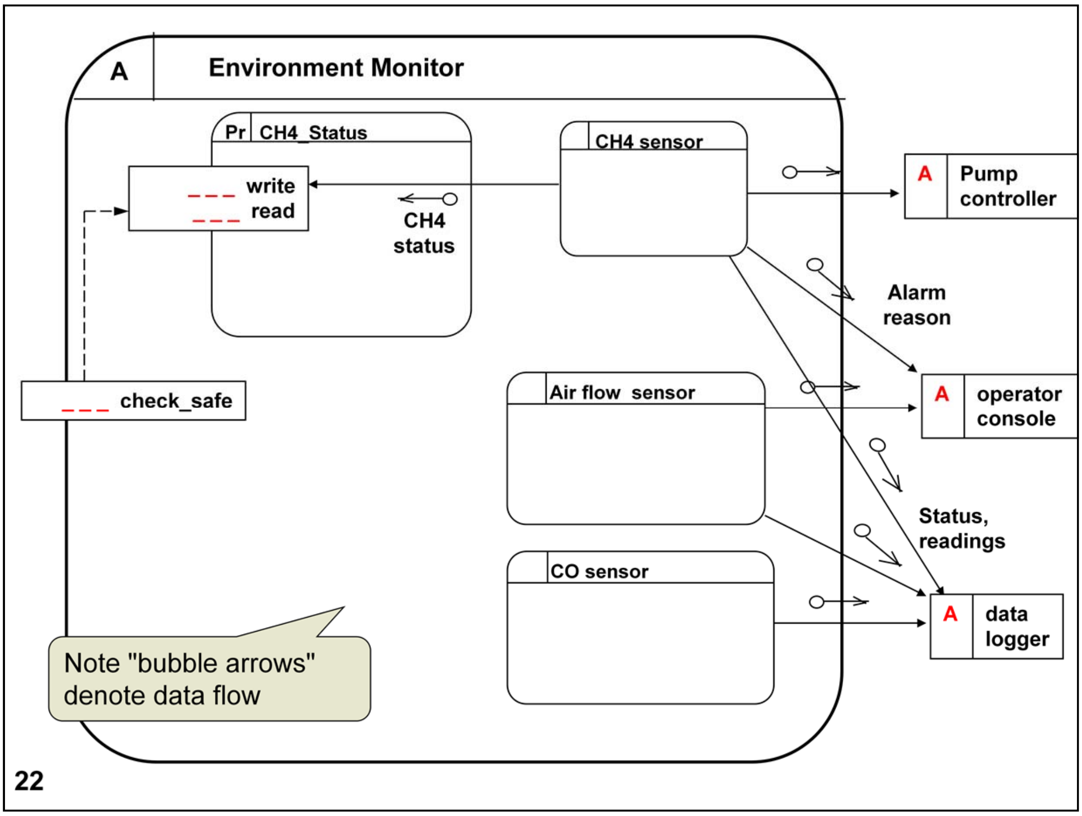


- Originally HRT-HOOD allowed any object type to be composite. At Swinburne we don't – it was too confusing.

- In our train system one terminal Active will be needed to use Swindows.Get_Char
- Swindows is an Environment object (ie provided), essentially a composite Active but not designed with HRT-HOOD.




- Shows major sub-systems
- Objects marked as A because we expect there will be threads
- Operation constraints not yet decided
- The authors chose poor names, so some have been changed
- **Set_Not_Safe**, **Set_safe** (orig is_safe) and Set_Pump are commands; request_status is a query.
- The Data_logger operations also have poor names
  - each should be a verb (or *verb_object*) but these look like *adjective_noun*, ie **should be** Log_CO etc

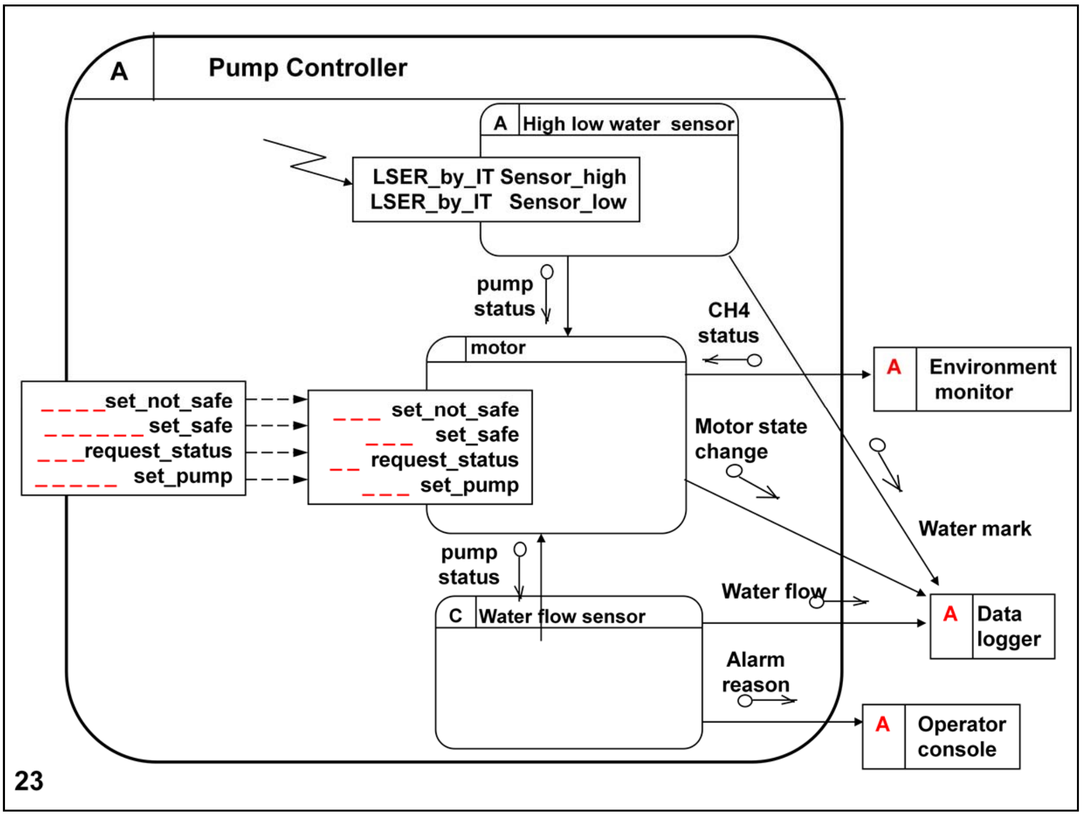



- Missing object types: all C
- Missing constraints:

  HSER check_safe -- appropriate constraint for an A object query **(2010: prefer PSER here)**

  PSER write -- normal for a Protected object

  PSER read -- " " "
- The protected object exists to support the query Check_Safe which is called by Motor (next slide).
  - Motor cannot call the cyclic CH4_Sensor directly because cyclics don't normally have operations.
    **[2012: In practice this design might be optimised, but this is a textbook example.]**
  - Also Motor needs an immediate response but when it calls the cyclic thread is probably asleep (blocked in a delay).
  - CH4_Sensor places the most recent value in CH4_Status.
  - Mutual exclusion is needed in case the value is complex (eg a record) – possibly an over-kill now but good design for future expansion.
- Note that we don't show the call from Motor here. By convention we don't show the clients in HRT-HOOD decomposition diagrams, just the calls outward.

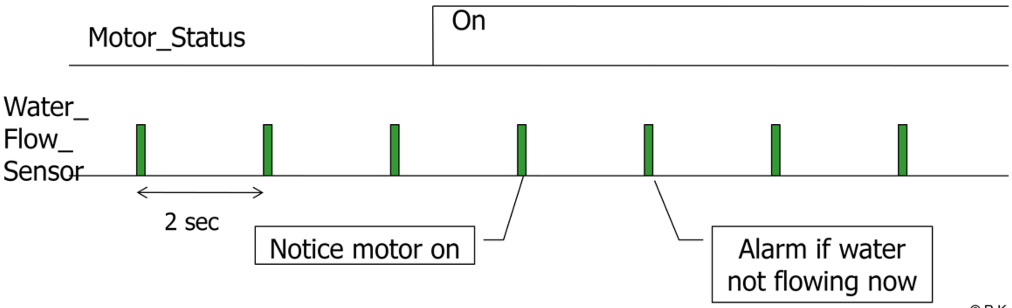



- What type for Motor?
    - Calls from outside.
    - Should it do things when it likes? No.
    - Do we need sporadic behaviour? No, in fact there are queries coming in.
    - Do we need mutex? Yes.
    - Does it call A objects? Yes.
    - Hence Motor is Pr but we'll have to be careful what it calls.
- Missing constraints:- motor ops: all PSER, so outer ops: **PSER**
- pump_status is essentially on or off. Whenever the motor is told to switch on it checks the CH4 level, ie whether danger of explosion.
- If the Environment monitor sees that CH4 is too high it calls operation Set_Not_Safe in case the motor is on.
  (This call not shown on this diagram -- we only see calls in which Pump_Controller is a client.)

*Real-Time Programming*
17/03/2012

# Notes: Water_Flow_Sensor

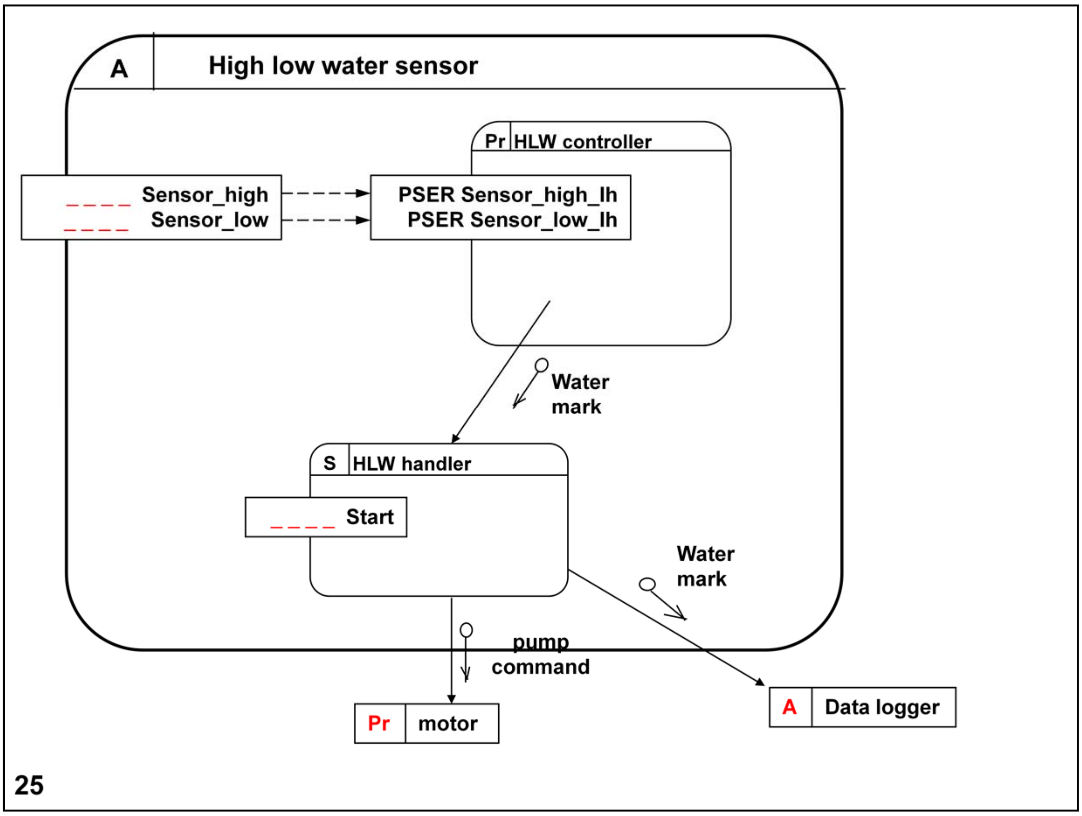
- This cyclic object checks every 2 seconds
  - whether water is flowing in the pipe, and
  - whether it should be flowing: by calling Motor.Request_Status
- It sends an alarm message to Operator_Console if the water flow doesn't start/stop within 2-4 seconds
  Example:




24



- The code has been simplified from that printed in Burns & Wellings 3rd ed -- see the zip on Blackboard.




- Missing constraints:

  LSER_by_IT Sensor_High -- appropriate for an A where an interrupt triggers a response. Such ops must be fast because during that time interrupts will be off!
  LSER_by_IT Sensor_Low

  ASER Start -- the normal op for an S

- In fact operations Sensor_High and Sensor_Low will not appear in the final Ada code because they will never be called. The two procedures of HLWController will be installed as interrupt handlers, ie called directly by the hardware. Interrupt handlers are normally Ada95 protected objects.

Real-Time Programming
2/04/2015

# Translation to Ada

- First/Second Level:
  - procedure Mine_Control_System
  - package Operator_Console
  - object Environment Monitor
    - packages CH4_Status, CH4_Sensor, …
  - object Pump Controller
    - packages Motor, Water_Flow_Sensor, HLW_Controller
  - package Data_Logger

Style 1:
Drop the outer packages

In this style we maintain the design hierarchy only in comments.

We also document the HRT-HOOD object types in comments, eg

```
-- PROTECTED part of Pump_Controller
package Motor is
```

26

© R.K.Allen,
Swinburne University of Technology

- (Read after working through the rest of this lecture:)
- What happened to package Pump_Controller? More generally, how do we maintain the HRT-HOOD design hierarchy in the code?
- Option 1 (this slide):
  comments, eg -- PROTECTED, part of Pump_Controller
- Option 2 (not shown, fairly obvious):
  include in the name, eg Pump_Motor, Pump_Water_Flow_Sensor
- Options 3, 4: next slide

*Real-Time Programming*
2/04/2015

## Translation to Ada : Child Packages

Recommended

- **Or** Style 2: use child packages
  - – object Environment Monitor becomes
    - packages Env, Env.CH4_Status, Env.CH4_Sensor, …
  - – object Pump Controller becomes
    - packages Pump, Pump.Motor, Pump.Water_Flow_Sensor, Pump.HLW_Controller

The parent may be empty and clients refer directly to the "provided" (ie public) child operations. File details:

| | |
|---|---|
| pump.ads | package Pump is<br>end Pump; |
| pump-motor.ads | package Pump.Motor is ← note hyphen vs dot<br>….<br>end Pump.Motor; |

27

© R.K.Allen,
Swinburne University of Technology

- Option 3 (recommended):
  use child packages with parents (possibly empty), no renames/call throughs eg:
  
  <as above>
- Option 4 (original ESA style without child packages) use renames , eg

```
with Motor, Water_Flow_Sensor, …
package Pump_Controller is
        procedure Not_Safe renames Motor.Not_Safe;
```

or call throughs:

```
package body Pump_Controller is
        procedure Not_Safe is
        begin Motor.Not_Safe;  end Not_Safe;
        …
```

BUT Motor itself has a call-through to the protected object – INEFFICIENT especially in programmers' time!

# Pump Motor Spec

```
package Pump.Motor is    -- PROTECTED

  type Pump_Status is (On, Off);
  -- … (other type declarations)

  Pump_Not_Safe : exception;    -- raised by Set_Pump

  procedure Set_Not_Safe;  -- set
  procedure Set_Safe;  -- set
  function Request_Status return Pump_Status;
  procedure Set_Pump(To : in Pump_Status);  -- on or off

end Pump.Motor;
```

These declarations are only here because they are needed by clients

See mine_case_study.zip





- Note the object's HRT-HOOD type is shown by a comment
- Booch icon shows the package spec (unshaded).
- The oval shows that this package declares Ada types as well as ops but it's not an ADT – perhaps the diagram would be clearer if the oval were omitted.

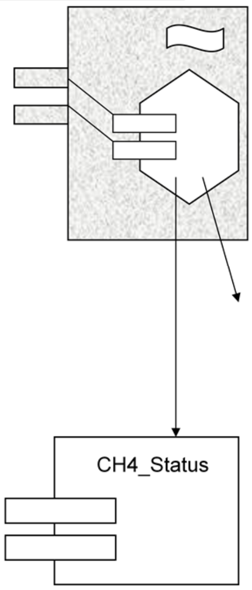
- The return type for Request_Status varied in the different editions of Burns & Wellings, sometimes Op_Status. The example Ada code mine_case_study.zip (2012) has Pump_Status.
- Other types declared in this spec are either no longer used or should be hidden in the package body.

*Real-Time Programming*

17/03/2012

## Pump Motor Body - 1

```
with Data_Logger, Env.CH4_Status, Device_Defs; …
package body Pump.Motor is

  Device_Reg : …

  protected Object is
          procedure Not_Safe;  -- set
          procedure Is_Safe …
  private
          Motor_Status : …
  end Object;

  procedure Set_Not_Safe is
  begin
          Object.Not_Safe;
  end Set_Not_Safe;
  …
```




**29**

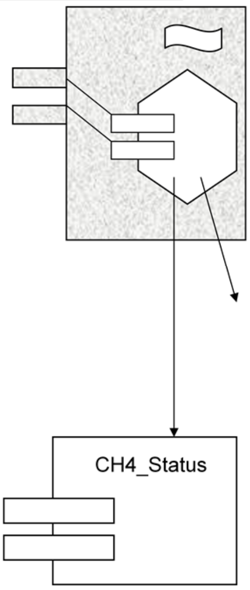


- Booch diagram: shows the package body of Motor (shaded) and the spec of CH4_Status. The “banner” shape represents data.
- Code:
  Note the ops visible in the package spec are ordinary procedures that “call through” to the corresponding ops (with similar name) belonging to the Ada protected.
- As usual declare the Ada protected first, then code things that call it and its body.
- “Object” is not a reserved word. Burns & Wellings originally used the name “OPSR” -- yuk! – then they used “Agent” but still no good as “agent” implies a thread.
- Here Motor_Status is protected. I recommend that Device_Reg be inside the protected. They couldn’t do that because they assumed memory-mapped IO – a representation clause specified the address of the variable. (Also they used name Pcsr instead of Device_Reg.)

Real-Time Programming
24/03/2015

# Pump Motor Body - 2

```
…

protected body Object is

      procedure Not_Safe is
      begin
         if …. -- lots of code
      end Not_Safe;

      …  -- other ops

end Object;

end Pump.Motor;
```




30

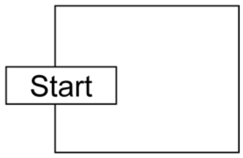


- Booch diagram: shows the package body of Motor (shaded) and the spec of CH4_Status. The “banner” shape represents data.
- Code:
  Note the ops visible in the package spec are ordinary procedures that “call through” to the corresponding ops (with similar name) belonging to the Ada protected.
- As usual have to declare the Ada protected first, then code things that call it and its body.
- “Object” is not a reserved word. Burns & Wellings originally used the name “OPSR” -- yuk! – then they used “Agent” but still no good as “agent” implies a thread.
- Here Motor_Status is protected. I recommend that Device_Reg is also inside the protected. They couldn’t do that because they assumed memory-mapped IO – a representation clause specified the address of the variable. (Also they used name Pcsr instead of Device_Reg.)

*Real-Time Programming*
26/03/2007

# HLW_Handler -- 1

```
• package Pump.Hlw_Handler is    -- SPORADIC
    type Water_Mark is (High, Low);

    procedure Start(Intr : Water_Mark);
    -- 2nd stage of interrupt (“intr”) handler
  end Hlw_Handler;
```




```
• with Data_Logger, Pump.Motor, Device_Defs; …
  package body Pump.Hlw_Handler is
    Hw_Control_Reg : …
    …
    procedure Sporadic_Code(Intr : Water_Mark) is
    begin
       if Intr = High then … else …
    end Sporadic_Code;
    procedure Initiallise is …
```

31



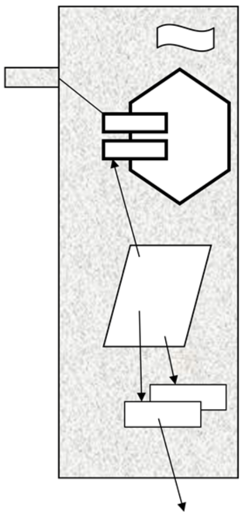
- This is the second stage of interrupt handling.
- Water_Mark declared here because it’s needed in calls to Start.

- The Booch diagram shows the package spec (not shaded).

***Real-Time Programming***
26/03/2007

## HLW Handler -- 2

```
task Thread;

protected Buffer is
        procedure Start(Intr : Water_Mark);
        entry Wait_Start(Intr : out Water_Mark);
 private
        W : Water_Mark; …
 end Buffer;

 procedure Start(Intr : Water_Mark) is
 begin
   Buffer.Start(Intr);
 end Start;

 protected body Buffer is ...
```



**32**



- Only a single item buffer and no over-run detection is needed so a Boolean has been omitted from the protected private part.
- Therefore the protected body is very simple – simpler than slide 12.
- Overrun code (and exceptions) omitted in this example to make it easier to follow

- The Booch diagram shows the package body (shaded).

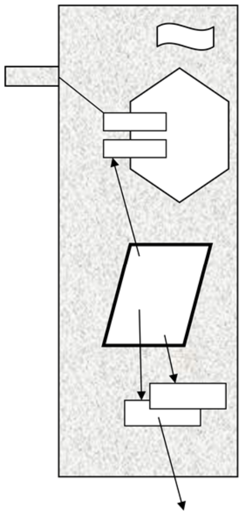
*Real-Time Programming*

26/03/2007

## HLW Handler - 3

```
task body Thread is
    Intr : Water_Mark;
begin
    Initiallise;
    loop
      Buffer.Wait_Start(Intr);
      Sporadic_Code(Intr);
    end loop;
end Thread;

end Pump.Hlw_Handler;
```

Note: no over-run code



33

© R.K.Allen,
Swinburne University of Technology

- Here the operation code is in procedure Sporadic_Code and this is outside the task. An alternative is to make it local to the task.
- The bottom arrow on the diagram represents calls by Sporadic_Code to Motor.Set_Pump and to Data_Logger.
- Overrun code (and exceptions) omitted in this example to make easier to follow