Real-Time Systems

Hermann Härtig

Real-Time Programming Languages

(ADA and Esterel as Examples)

TECHNISCHE UNIVERSITÄT DRESDEN

14/01/13

RT-HLL Issues

- Concurrency and Synchronization/Communication
- Time
  - Access to
  - Control over (“timeout”, ...)
- Scheduling/Resource Management
  - Built in
  - Explicit
- Recurrent processes

RT Language Classes

Synchronous HLL (tick driven)

- Esterel
- Lustre
- (State Charts)

Imperative HLL with rt-extensions

- ADA
- RT-Java
- PEARL
- CHILL
- RT-Euclid (designed to enable static analysis)

Esterel

For further study

Gerard Berry,
Esterel Language Primer
http://www.esterel-technologies.com/files/primer.zip

Video of Artist summer school 2008
http://www.artist-embedded.org/artist/Videos-Slides

Prof. Christian Hochberger's “Embedded Systems”

Caveat:

This lecture: introduction into principles only

Ignored: Extensive Tool Chain, Verification, ...

Synchronous Systems → Synchronous Languages

Gerard Berry:

> Our approach is based on the standard principle of separation of concerns. First, write the program at the logical level and make use of all the mathematics available there. Then, implement the program using the best available automatic synthesis tools, and check that the result is practically OK.

Ideal Systems produce their outputs synchronously with their inputs.

BUT: Implementation

Gerard Berry

For example, assume we want to trigger a reaction with two inputs A and B present.

We execute the following [compiler generated] C code:

```
M_I_A();

M_I_B();

M();
```

If X is emitted by the reaction, the function M() will call an output function M O X() we have to supply. The reaction is subject to an essential atomicity condition: during the execution of the main reaction function M(), it is forbidden to call any input function. This is the software counterpart to the hardware requirement that input voltages must be kept stable during a clock cycle.

Esterel at a glance

Most statements are instantaneous

(starts and terminates at the same instant of time)

Stepwise execution,

everything completes in each step/cycle/tick

Time consumption explicit (e.g., „Pause“)

Esterel: „Statements“

- Consume no time (unless explicitly said otherwise)
- Await A: „consumes one A“
- Pause: „consumes one time step“ (tick)

- X:= Y: assigns values to variables
- S1;S2
- S1 || S2
- Loop S end
  starts s, repeats if not terminated
  (s must consume time)

Esterel „Data“: Variables and Signals

Variable: Value

Signal: Value and Status

Value of any type

Status: Present/non present

Newly evaluated at every step

present when emitted

Signals

- Emit x(y): sets signal x present, assigns value y
- ?σ current value:
  value just emitted (if so) or value of previous instant (otherwise)
  pre(?S): previous value
- Present σ then s1 else s2 end (conditional)
- Abort S when σ do R end abort;
  starts S, terminates when σ becomes active, does R
- Suspend S when σ
  suspends S when σ active
  no emission when suspended
- Trap σ in S end trap
  starts S, aborts when σ present

Signals vs Variables

Gerard Berry

```
01   Emit Count(pre(?Count) + 1) vs V:= V+1;
```

Beware:

writing “emit COUNT(?COUNT+1)” is tempting but incorrect. Since ?COUNT is the current value of COUNT, it cannot be incremented and reemitted right away as itself. It is necessary to use the previous value pre(?COUNT).

More Statements

- halt: loop pause end
- await σ : abort halt when σ end abort
- sustain x(t): loop emit x(t); pause end
- loop S each R

  restarts S at each occurance of R

  every σ do S end every:

  await σ ; loop S each σ

Examples (all by Berry): ABRO

Gerard Berry

Specification ABRO:
Emit an output O as soon as two inputs A and B have occurred.
Reset this behavior each time the input R occurs.

module ABRO:

```
01   input A, B, R;
02   output O;
03   loop
04       [ await A || await B ];
05     emit O
06   each R
07   end module
```

Counting

Gerard Berry

Specification COUNT:
Count the number of occurrences of the input I seen so far, and broadcast it as the value of a COUNT signal at each new I.

module COUNT:

```
01   input I;
02   output COUNT := 0 : integer;
03   every I do
04     emit COUNT(pre(?COUNT) + 1)
05   end every
06   end module
```

Beware:
“emit COUNT(?COUNT+1)” is tempting but incorrect.

Speed

Gerard Berry

Specification SPEED:
Count the number of centimeters run per second, and broadcast that number as the value of a Speed signal every second.

module SPEED:

```
01   input Centimeter, Second;
02   relation Centimeter # Second;
03   output Speed : integer;
04   loop
05     var Distance := 0 : integer in
06       abort
07         every Centimeter do
08           Distance := Distance+1
09         end every
10       when Second do
11         emit Speed(Distance)
12       end abort
13     end var
14   end loop
15   end module
```

ADA

Used intensively, e.g. Military, Aircraft (B777), Space

- “most commonly used language in US weapons modernization”

Ada 83 - result of a competition ...

Ada 95 - major redesign (ISO/IEC 8652: 1995)

Ada 2005, includes Ravenscar: subset

Annex: Real-Time Systems

Few general points

Ada has “Annexes”:
    in this lecture: Real-Time Annex

Ada has “profiles”:
    relevant for this lecture “Ravenscar”
    reduced functionality for Hard-RT

Ada has “pragmas” (compiler directives)

CAVEAT:
    In this lecture: very limited extract relevant for RTS
    Especially, not covered explicitely:
    Packages, OO, Type-System, Generics, exceptions, ...
    we rely on your intuition

THE ADA-RT Book

Concurrent and Real-Time Programming in Ada

by

Alan Burns and **Andy Wellings**

Cambridge University Press

ISBN 978-0521866972

Most code examples taken from this source.

Many more resources available.

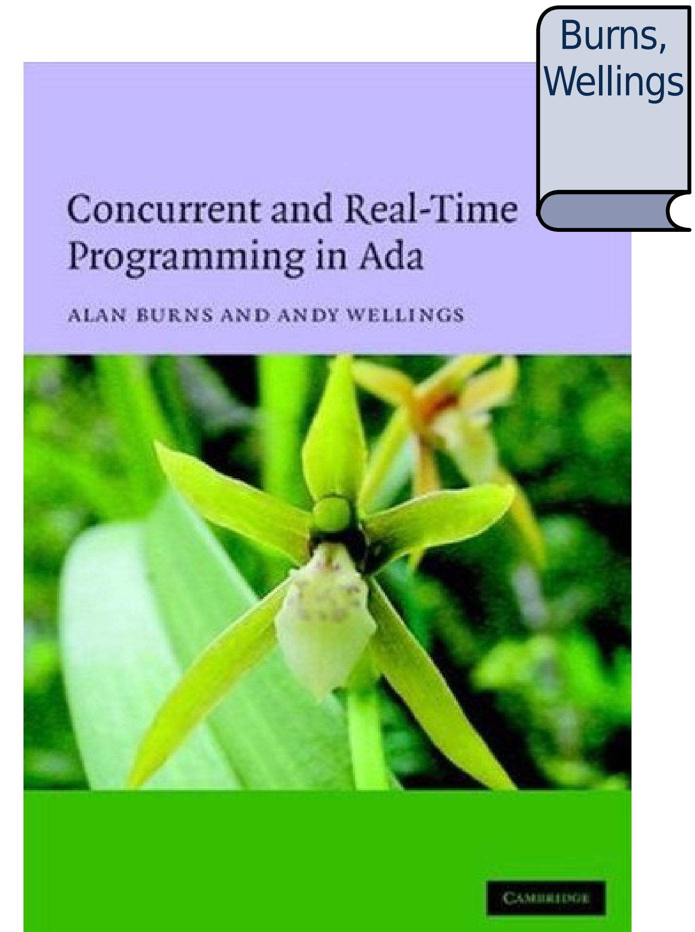


Image Source:
amazon.com/dp/B001GS6TBO/

Concurrency

Tasks

- are entities whose execution may proceed in parallel.
- have a thread of control.
- proceed independently,
  except at points where they synchronize.
- are created and activated via
  - an object declaration or
  - created dynamically using an “access type”
    Ptr:= new …

Example: Operator/Subscriber

Burns, Wellings
Ch. 5.1
Page 80

```
01   task type Subscriber;
02
03   task type Telephone_Operator is
04     entry Directory_Enquiry(Person : in Name; Addr : in Address;
05                             Num : out Number);
06   end Telephone_Operator;
07
08   S1, S2, S3 : Subscriber; -- friends of Stuart Jones
09   An_Op : Telephone_Operator;
10
11   task body Subscriber is
12     Stuarts_Number : Number;
13   begin
14     ...
15
16
17
18
19   end Subscriber;
```

Termination of Tasks

Every task has a “master” and “depends” on it:

block, subprogram etc.
containing the declaration of the task object or
of the access object type

Before leaving the master, the parent task waits for all dependent tasks to terminate.

Communication

- Protected objects

  for synchronized access to shared data

- Rendezvous

  for synchronous communication between tasks

- Unprotected access to shared data (global variables)

The Rendezvous

Based on client/server model:

- One task (client) calls an entry of an other task
- Other task accepts a call (Server)
- Calling task placed on a queue

Example: Operator/Subscriber

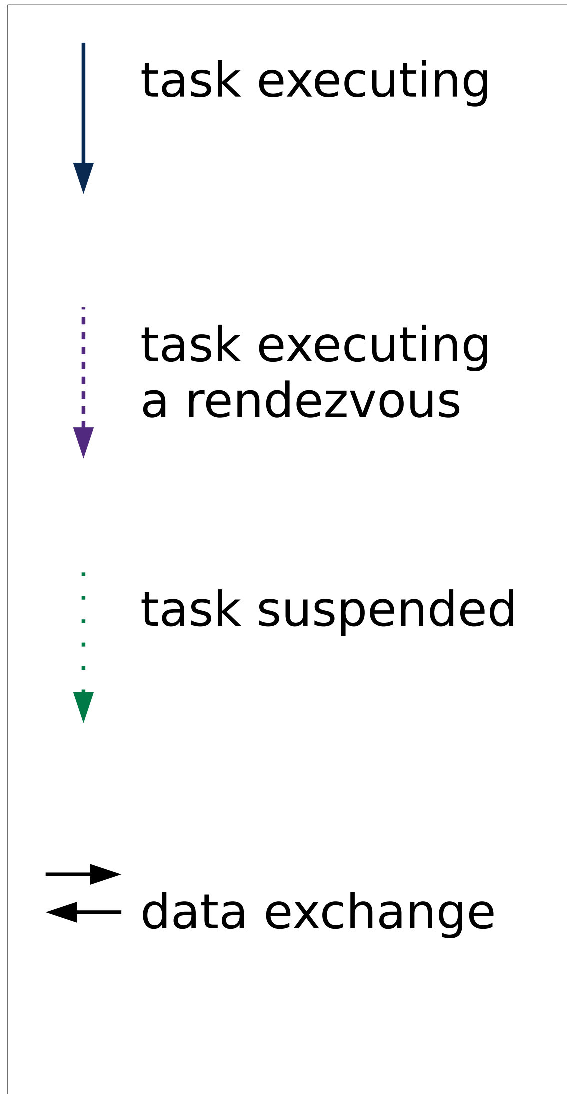
Burns, Wellings Ch. 5.1 Page 80 f.

```
01   task body Subscriber is
02     Stuarts_Number : Number;
03   begin
04   --.AD  ...
05     An_Op.Directory_Enquiry("STUART JONES",
06       "10 MAIN STREET, YORK", Stuarts_Number);
07     -- phone Stuart
08   --.AD  ...
09   end Subscriber;
```

```
01    task body Telephone_Operator is
02    begin
03      loop
04        -- prepare to accept next call
05        accept Directory_Enquiry(Person : in Name;
06               Addr : in Address; Num : out Number) do
07          -- look up telephone number and
08          -- assign the value to Num
09          null; --.RM
10        end Directory_Enquiry;
11        -- undertake housekeeping such as logging all calls
12      end loop;
13    end Telephone_Operator;
```

In detail

Burns, Wellings
Ch. 5.1
Page 82



Select Statement

A task can have multiple entries:

```
01  select
02     when (expression) =>
03     accept E1 do  bla end E1;
04  or
05     when (expression) =>
06     accept E2 do  bla end E2;
07  or ...
08  end select;
09
```

Select Statement

- Arbitrary entry, whose expression is evaluated to true, is called.
- Exception if no expression evaluates to true.
- The Boolean expression is evaluated only once per execution of select
  (do not use global variables in when clause !)

More on “select” later (timings)

Protected Objects

have:

- functions
  - read-only access to shared data
  - functions can be executed concurrently
- procedures
  - read/write access to shared data
  - Procedures: executed under mutual exclusion
- entries
  - procedures with barriers (boolean expressions)
  - synchronization operations automatic
  - evaluate barriers
    at each call to and at each return from an entry

Example: Resource with Locking

```
01  protected type Resource is
02      entry      lock;
03      procedure  unlock; Release;
04      private    Locked: Boolean := False;
05  end Resource;
06
07  protected body Resource is
08      entry lock when not Locked is
09      begin
10        Locked := True;
11      end lock;
12
13      procedure unlock is begin  Locked := False end unlock;
14  end Resource;
```

Tasks ./. Protected Objects

Burns, Wellings
Ch. 7.2
Page 131 f.

```
01   protected type Shared_Integer(Initial_Value : Integer) is
02     function Read return Integer;
03     procedure Write(New_Value : Integer);
04     procedure Increment(By : Integer);
05   private
06     The_Data : Integer := Initial_Value;
07   end Shared_Integer;
```

```
01   protected body Shared_Integer is
02     function Read return Integer is
03     begin
04       return The_Data;
05     end Read;
06
07     procedure Write(New_Value : Integer) is
08     begin
09       The_Data := New_Value;
10     end Write;
11
12     procedure Increment(By : Integer) is
13     begin
14       The_Data := The_Data + By;
15     end Increment;
16   end Shared_Integer;
```

Tasks ./. Protected Objects

Burns, Wellings Ch. 7.2 Page 132

```
01   task body Shared_Integer is
02     The_Data : Integer := Initial_Value;
03   begin
04     loop
05       select
06         accept Read(New_Value : out Integer) do
07           New_Value := The_Data;
08         end Read;
09       or
10         accept Write(New_Value : Integer) do
11           The_Data := New_Value;
12         end Write;
13       or
14         accept Increment(By : Integer) do
15           The_Data := The_Data + By;
16         end Increment;
17       or
18         terminate;
19       end select;
20     end loop;
21   end Shared_Integer;
```

Priorities

Burns, Wellings Ch. 13.2 Page 320

```
01  pragma task_dispatching_policy(policy identifier);
```

Supported by ADA 205:

- Preemptive fixed priority
- Non-Preemptive fixed priority
- Round robin
- EDF
- And mixtures thereof

Fixed Priorities

- `task (type) T is   pragma Priority(P);  ....`
- Distinct Run Queue per (active) priority

  released tasks at the end of queue

  preempted tasks at the beginning
- Such priorities are called the *base* priorities of the task

  in contrast to *active* priority

Priority Ceiling

Burns, Wellings
Ch. 13.3
Page 325

```
01  Pragma Locking_Policy(Ceiling_Locking);
02  Protected object is pragma priority(...) ... ;
```

Implements the “immediate ceiling protocol”

- The object ceiling priority must be maximum priority of any calling task
- the task executing a protected operation executes at the ceiling priority of the protected object

Active Priority

Base priority or

- Ceiling priority if calling a protected object
- The creating task's priority if higher than the base p
- During rendezvous:

  the priority of the task executing the accept statement

  inherits the priority of the calling task

Run-Time Priorities

```
01   procedure Set_Priority(Priority: Any_Priority;
02                          T: Task_ID := Current_Task);
03
04
05   function Get_Priority(T: Task_ID := Current_Task)
06                          return Any_Priority;
07
```

Resets base priority.

EDF Dispatching

Burns, Wellings
Ch. 14.3
Page 336 f.

How to set deadlines:

```
01   Package ada.dispatching.EDF is
02   procedure set_deadline (D: in deadline,
03                           T: in TaskId)
04   Procedure DelayUntilAndSetDeadline(...)
05   Procedure GetDeadline (...)
```

Or

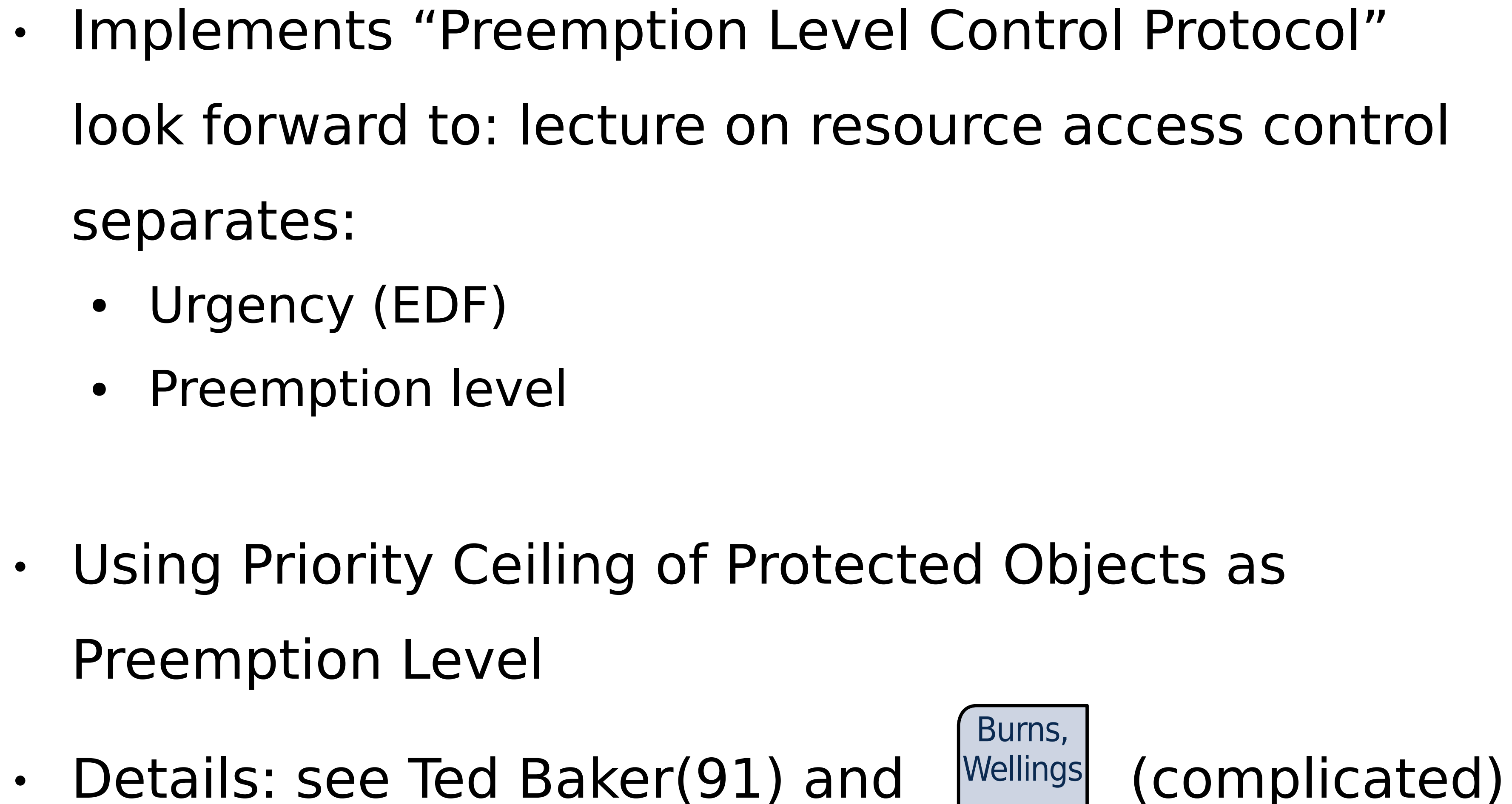
```
01   Pragma Relative_Deadline(Milliseconds(3))
02   & Explicit call to set first deadline of periodic task
```

EDF and Ceiling

- Implements “Preemption Level Control Protocol”

  look forward to: lecture on resource access control

  separates:
  - Urgency (EDF)
  - Preemption level

- Using Priority Ceiling of Protected Objects as Preemption Level
- Details: see Ted Baker(91) and 

(complicated)

Mixed Scheduling Policies, example

Burns, Wellings Ch. 14.4 Page 347

Specify “priority partitions” to set scheduling disciplines

```
01   Pragma Priority_Specific_Dispatching
02   (Fifo_Within_Priorities, 10, 16)
03
04
05   Pragma Priority_Specific_Dispatching
06   (EDF_Across_Priorities, 2,9)
07
08
09   Pragma Priority_Specific_Dispatching
10   (RoundRobin_Within_Priorities, 1, 1)
```

Entry Queuing: Implicit Policies

Burns, Wellings
Ch. 13.4
Page 327

Standard: FIFO

```
01   pragma Queuing_Policy(Priority_Queuing);
```

> "the user can override the default FIFO policy with the pragma Queuing_Policy"

per partition (not per entries or tasks)

passing of dynamic priorities as implicit parameters

Explicit Request Ordering: Requeue

Explicit treatment of request orders, example:

- A request enters entry or barrier
- Parameters inspected in body code
- Possible decision: requeue at different entry

Action:

- Requeue a request of a caller to some entry or barrier

Not easy to use !!!

Example

```
01  protected body AirportGate is
02      entry EnterGateBusiness(Ticket)
03      begin
04          if Ticket.Economy then
05           requeue EnterGateEconomy;
06          end if;
07          HandleBusinessPassenger
08      end EnterGateBusiness;
09
10      entry EnterGateEconomy(Ticket)
11      when AllBusinessPassengersHaveEntered
12      begin HandleEcenomyPassenger end EnterGateEconomy
13  end AirportGate;
14
```

Timing events

Burns, Wellings Ch. 15.2 Page 363

```
01   Package Ada.Real_time.Timing_events is
02   --...
03
04   Procedure Set_Handler(Event: in out Timing_Event;
05   At-Time: Time; Handler: Timing_Event_Handler);
06   --...
07   Procedure Set_Handler(Event: in out Timing_Event;
08   At-Time: Time_Span; Handler: Timing_Event_Handler);
```

Causes Handler to be called at chosen times.

Handlers are called by Clock_Interrupt Handler

Must not block.

Used for periodic action and watchdogs

Example: Watchdog

Burns, Wellings
Ch. 15.2
Page 364 f.

```
01  protected Watchdog is
02    pragma Interrupt_Priority (Interrupt_Priority'Last);
03    entry Alarm_Control;
04       -- Called by alarm handling task.
05    procedure Call_In;
06       -- Called by application code every 50ms if alive.
07    procedure Timer(Event : in out Timing_Event);
08       -- Timer event code, ie the handler.
09  private
10    Alarm : Boolean := False;
11  end Watchdog;
12
13  Fifty_Mil_Event : aliased Timing_Event;
14  TS : Time_Span := Milliseconds(50);
15
16  Set_Handler(Fifty_Mil_Event, TS, Timer);
```

Example: Watchdog

Burns, Wellings
Ch. 15.2
Page 365

```
01  protected body Watchdog is
02    entry Alarm_Control when Alarm is
03    begin
04      Alarm := False;
05    end Alarm_Control;
06
07    procedure Timer(Event : in out Timing_Event) is
08    begin
09      Alarm := True;
10      -- Note no use is made of the parameter in this example
11    end Timer;
12
13    procedure Call_in is
14    begin
15      Set_Handler(Fifty_Mil_Event, TS, Timer);
16      -- This call to Set_Handler cancels the previous call
17    end Call_in;
18  end Watchdog;
```

Execution Time Timers

Burns, Wellings
Ch. 15.8
Page 372

```
01  Package Ada.Execution_Time.Timers is
02  --...
03
04  Procedure SetHandler(TM : in out Timer;
05  In_Time: Time_Span; Handler. Timer_Handler);
06
07  Procedure SetHandler(TM : in out Timer;
08  At_Time: CPU_Time; Handler. Timer_Handler);
09  --...
10
```

Timers are strongly linked to tasks!

Time: Delay Statement

```
01  delay
02     duration
03     point in time
04
05  delay 5.0;           -- delay for at least 5 seconds
06  delay until A_Time; -- delay at least until A_Time
```

specifies minimum delay

Delay and Select, server side

```
01   select
02       accept An_Entry do  bla
03       end An_Entry;
04   or
05       delay 10.0;
06       Put("An_Entry: timeout");
07   end select;
```

Select terminates if entry is not called within 10 time units.

Delay and Select, client side(1)

```
01   select
02       Operator.Enquiry()
03   or
04       delay 10;
05   end select;
```

Select terminates if entry is not accepted within 10 time units.

Only one call alternative allowed

Client side (2): “Asynchronous” Select

Burns, Wellings Ch. 9.3 Page 202

```
01  select trigger
02
03  triggering_alternative --- (entry-call or delay)
04
05  then abort
06
07  abortable_part
08
09  end select;
10
```

- If delay or entry-call complete before the abortable part, the abortable part is aborted
- abortable_part must not an accept statement

Example

```
01   select
02       delay 5.0;                -- triggering alternative
03   then abort
04       CalculationComplete:= false;
05       Invert_Giant_Matrix(M);   -- abortable part
06       CalculationComplete:= true;
07   end select;
```

Careful: notice the race condition !

Example: Watchdog

Burns, Wellings Ch. 6.3 Page 104

```
01  task type Watchdog is
02    entry All_Is_Well;
03  end Watchdog;
04
05  task body Watchdog is
06  begin
07    loop
08      select
09        accept All_Is_Well;
10      or
11        delay 10.0;
12        -- signal alarm, potentially the client has failed
13        exit;
14      end select;
15    end loop;
16    -- any further required action
17  end Watchdog;
```

Example: Operator/Subscriber

Burns, Wellings
Ch. 6.1
Page 99

```
01   task type Telephone_Operator is
02     entry Directory_Enquiry(Person : in Name; Addr: in Adress;
03                             Num : out Number);
04     entry Directory_Enquiry(Person : in Name;
05           Zip : in Postal_Code; Num : out Number);
06     entry Report_Fault(Num : Number);
07   private
08     entry Allocate_Repair_Worker(Num : out Number);
09   end Telephone Operator;
```

Example: Operator/Subscriber

Burns, Wellings
Ch. 6.1
Page 99 f.

```
01  task body Telephone_Operator is
02    Workers : constant Integer := 10;
03    Failed : Number;
04  begin
05    loop
06      -- prepare to accept next request
07      select
08        accept Directory_Enquiry(Person : in Name;
09           Addr : in Adress; Num : out Number) do
10           -- look up telephone number and assign the value to Num
11        end Directory_Enquiry;
12      or
13        accept Directory_Enquiry(Person : in Name;
14           Zip : in Postal_Code; Num : out Number) do
15           -- look up telephone number and assign the value to Num
16        end Directory_Enquiry;
17      or
18        accept Report_Fault(Num : Number) do
19           Failed := Num;
20        end Report_Fault;
21           -- log faulty line and allocate repair worker
22      end select; – undertake housekeeping such as logging all calls
23    end loop;
24  end Telephone_Operator;
```

Example: Operator/Subscriber

Burns, Wellings
Ch. 6.9
Page 119

```
01  task type Subscriber;
02
03  task body Subscriber is
04    Stuarts_Number : Number;
05  begin
06    loop
07      --...
08      select
09        An_Op.Directory_Enquiry("STUART JONES",
10              "10 MAIN STREET, YORK", Stuarts_Number);
11        -- log the cost of a directory enquiry call
12      or
13        delay 10.0;
14        -- phone up his parents and ask them,
15        -- log the cost of a long distance call
16      end select;
17      --...
18    end loop;
19  end Subscriber;
```

Simple Periodic Task With Static Priority

Burns, Wellings Ch. 14.3 Page 345

```
01  task A is
02    pragma Priority(5);
03  end A;
04
05  task body A is
06    Next_Release: Real_Time.Time;
07  begin
08    Next_Release := Real_Time.Clock;
09    loop
10      -- code
11      Next_Release := Next_Release + Real_Time.Milliseconds(10);
12      delay until Next_Release;
13    end loop
14  end A;
```

Recurrent Tasks (1)

Burns, Wellings

```
01  with Ada.Task_Identification; use Ada.Task_Identification;
02  with Ada.Real_Time; use Ada.Real_Time;
03  package Periodic_Scheduler is
04    procedure Set_Characteristic(T : Task_Id; Period : Time_Span;
05                                 First_Schedule : Time);
06    procedure Wait_Until_Next_Schedule; -- potentially blocking
07  end Periodic_Scheduler;
08
09  with Ada.Task_Attributes;
10  package body Periodic_Scheduler is
11    Start_Up_Time : Time := Clock;
12    type Task_Information is
13      record
14        Period : Time_Span := Time_Span_Zero;
15        Next_Schedule_Time : Time :=
16                      Time_Of(100_000,Time_Span_Zero);
17  end record;
18    Default : Task_Information;
19        -- a default object needs to be provided
20        -- to the following package instantiation
21
22    package Periodic_Attributes is new
23          Ada.Task_Attributes(Task_Information, Default);
24    use Periodic_Attributes;
```

Recurrent Tasks (2)

Burns, Wellings
Ch. 12.4
Page 315

```
25      procedure Set_Characteristic(T : Task_Id; Period : Time_Span;
26                                   First_Schedule : Time) is
27      begin
28        Set_Value((Period, First_Schedule), T );
29      end Set_Characteristic;
30
31      procedure Wait_Until_Next_Schedule is
32        Task_Info : Task_Information := Value;
33        Next_Time : Time;
34      begin
35           Next_Time := Task_Info.Period +
36                Task_Info.Next_Schedule_Time;
37           Set_Value((Task_Info.Period,Next_Time));
38          delay until Next_Time;
39      end Wait_Until_Next_Schedule;
40    end Periodic_Scheduler;
```

Recurrent Tasks (3)

Burns, Wellings

Periodic tasks can now be encoded as

```
01    task Periodic_Task;
02    task body Periodic_Task is
03
04    begin
05      loop
06        -- statements to be executed each period
07          Periodic_Scheduler.Wait_Until_Next_Schedule
08      end loop;
09    end Periodic_Task;
```

Example EDF

Burns, Wellings Ch. 14.3 Page 345

```
01  task A is
02    pragma Priority(5);
03    pragma Relative_Deadline(10);
04        -- gives an initial relative
05        -- deadline of 10 milliseconds
06  end A;
07
08  task body A is
09    Next_Release: Real_Time.Time;
10  begin
11    Next_Release := Real_Time.Clock;
12    loop
13      -- code
14      Next_Release := Next_Release + Real_Time.Milliseconds(10);
15      Delay_Until_And_Set_Deadline(Next_Release,
16                                   Real_Time.Milliseconds(10);
17    end loop
18  end A;
```

Finally the dispatching policy must be changed from

```
01  pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

to

```
01  pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

Example

Burns, Wellings Ch. 15.5 Page 373

```
01    protected Overrun is
02      pragma Priority(Min_Handler_Ceiling);
03      entry Stop_Task;
04      procedure Handler(TM : in out Timer);
05      procedure Reset(C1, C2 : Time_Span);
06    private
07      Abandon : Boolean := False;
08      First_Occurence : Boolean := True;
09      WCET : Time_Span;
10      WCET_Overrun : Time_Span;
11    end Overrun;
```

Example

Burns, Wellings Ch. 15.5 Page 373

```
01    protected body Overrun is
02      entry Stop_Task when Abandon is
03      begin
04        Abandon := False;
05        First_Occurence := True;
06      end Stop_Task;
07      procedure Reset(C1, C2 : Time_Span) is
08      begin
09        Abandon := False;
10        First_Occurence := True;
11        WCET := C1;
12        WCET_Overrun := C2;
13      end Reset;
14      procedure Handler(TM : in out Timer) is
15      begin
16        if First_Occurence then
17          Set_Handler(TM,WCET_Overrun,Handler);
18          Set_Priority(2, TM.T.all);
19          First_Occurence := False;
20        else
21          Abandon := True;
22        end if;
23      end Handler;
24    end Overrun;
```

Example

Burns, Wellings
Ch. 15.5
Page 373

```
01  task Hard_Example;
02
03  task body Hard_Example is
04    ID : aliased Task_ID := Current_Task;
05    WCET_Error : Timer(ID);
06    WCET : Time_Span := Microseconds(1250);
07    WCET_Overrun : Time_Span := Microseconds(250);
08    Bool : Boolean := False;
09    ...
```

Example

Burns, Wellings
Ch. 15.5
Page 373 f.

```
10  begin
11    -- initialisation
12    loop
13      Overrun.Reset(WCET, WCET_Overrun);
14      Set_Handler(WCET_Error,WCET,Overrun.Handler);
15      select
16        Overrun.Stop_Task;
17        -- handle the error if possible
18      then abort
19        -- code of the application
20        null; --.RM
21      end select;
22      Cancel_Handler(WCET_Error, Bool);
23      Set_Priority(14);
24     delay until ...
25    end loop;
26   ...
27  end Hard_Example;
```

Missing in this RT-HLL lecture

- RT-Java and RT-Garbage Collection
- Language with built-in periodic processes

To take away ...

- Principles of synchronous languages
- Mechanisms to explicitely handle timing
- Mechanisms to handle asynchronous events
- “scheduling” of processes, queues, ...