# Programming Real-Time Embedded systems : Ada 2005 and RTEMS

Frank Singhoff

Office C-202

University of Brest, France

singhoff@univ-brest.fr

# Summary

1. Introduction and sequential programming.
2. Concurrency features.
3. Real-Time features.
4. Examples of Ada runtimes.
5. Conclusion.
6. References.

# Introduction to Ada (1)

- **Why this language for Real-Time embedded systems:**
  - Has concurrency and real-time features: task, interruptions, synchronization, timers, real-time scheduling features,...
  - International ISO standard (portability).
  - Separate compilation (large software).
  - Provides numerous reliability mechanisms (e.g. strong typing).
  - Complex language.

- **Domains:** transportations (train, aircraft, spacecraft) and military devices.

- **Examples:** Airbus (320, 380), Boeing (777), Fokker, Tupolev, Eurostar, Underground (line 14 of Paris), TGV, Ariane (4 and 5), Satellites (Intersat), spacecraft (Cassini, Huygens, Soho, Mars Express), military (Tiger, Apache, Patriot) $\Longrightarrow$ http://www.seas.gwu.edu/ mfeldman/ada-project-summary.html.

- **Historical matters:** Ada 83, Ada 95, Ada 2005, Ada 2012.

# Introduction to Ada (2)

1. What is an Ada program?
2. Types, operators, variables, constants.
3. Flow of control.
4. Inputs/Outputs.
5. Pointers and dynamic allocations.
6. Generic packages.

# What is an Ada program (1)

- **Separate compilation:** unit of program = unit of compilation.

- **Types of program units (GNAT files) :**
  - **Main procedures:** entry point of a program (.adb file).
  - **Packages:** set of declarations (subprograms, types, tasks, ...).
    - Public part: (package specification, .ads file).
    - Hidden/private part: (package body, .adb file).
  - **Tasks :** specification (.ads file) and body (.adb file).
  - **Generic units:** units that are parametrized (packages or subprograms). Possible parameters: types, constants, subprograms or packages.

# What is an Ada program (2)

- **Structure of a main procedure:**

```
with package_name1; use package_name1;
with package_name2; use package_name2;

procedure main_procedure_name is
-- declarations
begin
-- statements
end main_procedure_name;
```

- File `main_procedure_name.adb`
- `with` and `use` clauses.
- Use is optional (pointed notation otherwise) $\Longrightarrow$ do not use them for reliable software.

# What is an Ada program (3)

- **Example of a main procedure:**

```
with text_io;
use text_io;

procedure Hello is
begin
   Put_Line("Hello world");
end Hello;
```

# What is an Ada program (4)

- **Structure of a package specification:**

```
package package_name is
-- public declarations
private
-- private declarations
end package_name;
```

- **Structure of a package implementation:**

```
package body package_name is
-- sub-programs
begin
-- initialization of the package
end package_name;
```

# What is an Ada program (5)

- **Package specification (file `mypackage.ads`) :**

```
package mypackage is

   procedure sum(a : in integer;
      b : in integer; result : out integer);
   function sum(a : in integer; b : in integer)
      return integer;

private
   internal_variable : integer;
end mypackage;
```

# What is an Ada program (6)

- **Package implementation (`mypackage.adb`) :**

```
package body mypackage is

   procedure sum(a : in integer;
      b : in integer; result : out integer) is
   begin
      result:=a+b+internal_variable;
   end sum;
   function sum(a : in integer; b : in integer)
      return integer is
   begin
      return a+b+internal_variable;
   end sum;

begin
   internal_variable:=100;
end mypackage;
```

# What is an Ada program (7)

- **Use of `mypackage` :**

```
with text_io;
use text_io;
with mypackage;
use mypackage;

procedure main is
a : integer :=0;
begin
   sum(10,20,a);
   put_line(integer'image(a));
   a:=sum(40,50);
   put_line(integer'image(a));
end main;
```

# What is an Ada program (8)

- **Compile this program (with GNAT) :**

```
>gnatmake main.adb
gcc -c mypackage.adb
gcc -c main.adb
gnatbind -x main.ali
gnatlink main.ali
```

- **gnatmake** : manage compilation unit dependencies.
- **gcc** : compile.
- **gnatbind** : elaboration (packages initialization).
- **gnatlink** : addresses link edit.
- **Results** : `main, mypackage.ali, mypackage.o, main.ali` and `main.o`

# What is an Ada program (9)

- **Exercise 1:**

```
package Compute is
    function Add(A : in Integer; B : in Integer)
            return Integer;
    function Multiply(A : in Integer; B : in Integer)
            return Integer;
    function Substract(A : in Integer; B : in Integer)
            return Integer;
    function Divide(A : in Integer; B : in Integer)
            return Integer;
end Compute;
```

Write a main procedure which computes and displays the value of the following expression: $(2 \cdot 3) + 4$. Provide also a package implementation for `compute`.

# Types, operators, variables (1)

- **Strong typing:**
  - Increase maintainability and source code readability.
  - Increase safety: static analysis at compilation time, runtime time exception => reduce latency of bug occurrence.
  - Forbid operation between variables with different types (no implicit cast).
- **What is a type:**
  - Type = size in memory and its representation + allowed possible values + attributes/operators.
  - Range of possible values defined by the standard (portability).
  - Attributes : pre-defined operators for any types (including the types you may define).

# Types, operators, variables (2)

- **Scalar types:**
  - `float, integer, boolean, character, access` and enumerations.
  - Examples of attributes: `integer'last, integer'first, integer'range`

- **Composed types:** `array, string` (which is also an array), `record, union, task, protected`

- **Main operators:**
  - Arithmetic: `+, -, *, /, mod`
  - Relational: `=, /=, <=, >=, in, not, and, or, xor`

# Types, operators, variables (3)

- **Derived types:** if type $a$ is derived from type $b$, then $a$ and $b$ are two different types that are non compatible.
- **Subtypes:** if type $a$ is a subtype of type de $b$, then $a$ and $b$ are compatible. $a$ is an alias of $b$.

# Types, operators, variables (4)

- **Examples of declarations:**

```
with Text_io;
use Text_io;

procedure Declare_Var is
   i1 : Integer;
   i2 : Integer := 0;
   s1 : String (1..10);
   f1 : Constant Float := 10.5;
begin
   Put_Line("Integer'First=" & Integer'Image(Integer'First));
   Put_Line("Integer'Last=" & Integer'Image(Integer'Last));
end Declare_Var;
```

# Types, operators, variables (5)

- **Some subtypes and derived types:**

```
procedure Derive is
   type Temperature is new Integer Range –280 .. 300;

   t1 : Temperature := 0;
   t2 : Temperature := 300;
   i  : Integer :=10;

Begin
   t1:=t1+t2;
   t1:=t1+i;
   t2:=t2+1;
end Derive;
```

# Types, operators, variables (6)

- **Some subtypes and derived types:**

```
procedure Derive is
   subtype Temperature is Integer Range –280 .. 300;

   t1 : Temperature := 0;
   t2 : Temperature := 300;
   i  : Integer :=10;

Begin
   t1:=t1+t2;
   t1:=t1+i;
   t2:=t2+1;
end Derive;
```

# Types, operators, variables (7)

- Strong typing allows static analysis.
- Example (from D. Lesens [LES 10]):

```
// Wrong C program ...
// but this program will compile !
typedef enum {ok, nok} t_ok_nok;
typedef enum {off, on} t_on_off;

void main() {
   t_ok_nok status = nok;
   if (status == on)
      printf("is on\n");
}
```

# Types, operators, variables (8)

- And the Ada program now:

```
with Text_Io;
use Text_Io;

-- Wrong Ada program ...
-- but this program will not compile

procedure Ada_Wrong is
   type t_ok_nok is (ok, nok);
   type t_on_off is (off, on);
   status : t_ok_nok := nok;
begin
   if (status = on)
      then Put_Line("is on\n");
   end if;
end Ada_Wrong;
```

# Types, operators, variables (9)

- **Composed types:**

1. **Type constructor:** `type`
2. **Enumeration:** discrete type, memory representation is hidden (similar to C `enum`) but specific attributes (`succ` and `pos`).
3. **Record:** set of variables (similar to C struct). Initialization of each field can be done either by declaration order or by giving its name.
4. **Array:** 1 or 2 dimensions. Indexes must be discrete types (integer or enumeration). Array size is known at type declaration (constrained array) or at variable declaration (unconstrained array).

# Types, operators, variables (10)

- **Example of an enumeration:**

```
with text_io;
use text_io;
procedure enumeration is

type a_day is (monday, tuesday, wednesday, thuesday,
  friday, saturday, sunday);
j : a_day := monday;

package io is new text_io.enumeration_io(a_day);

begin
  io.Put(a_day'first);
  io.Put(a_day'last);
  j:=a_day'succ(j);
  io.Put(j); Put_Line(a_day'image(j));
end enumeration;
```

# Types, operators, variables (11)

- **Example of an array:**

```
type a_day is (monday, tuesday, wednesday, thursday,
     friday, saturday, sunday);
type tab1 is array (0..3) of integer;
type tab2 is array (1..4) of a_day;
type tab3 is array (monday..sunday) of integer;

t1 : tab1 := (30,43,28,100);
t2 : tab2 := (4=>monday, 2=>tuesday,
   3=>sunday, 1=>wednesday);
t3 : tab3;

begin
  t1(0):=t1(0)*2;
  t2(2):=monday;
  t3(monday):=2;
  ...
```

# Types, operators, variables (12)

- **Example of a record:**

```
with Text_Io;
use Text_Io;
procedure Point is

type A_Point is record
    X : Integer;
    Y : Integer;
end record;

P1 : A_Point := (10,20);
P2 : A_Point := (Y=>20, X=>10);

begin
   Put_Line(Integer'Image(P1.X));
   Put_Line(Integer'Image(P1.Y));
end Point;
```

# Types, operators, variables (13)

- **Exercise 2 :** for each following statement, check if it compiles or not. Explain why it can not be compiled.

```
type t1 is new integer range 0..10;
type t2 is new integer range 0..100;
subtype t3 is t1;
subtype t4 is t3;
subtype t5 is t2;

a, b : t1;
c : t2;
d : t3;
e, f : t4;

a:=b+c;
d:=c*a;
d:=c*f;
f:=a+b;
e:=e*100;
```

# Flow of control (1)

- Sequence:

```
i1 ; i2
```

- Conditional test:

```
if cond
   then i1 ;
   else i2 ;
end if ;
```

# Flow of control (2)

- Some different loops:

```
while cond
  loop
        i1; i2;
end loop;

for i in a..b loop
    i1; i2;
end loop;

-- Typical real-time design
loop
    i1; i2;
    exit when cond;  -- optional test
end loop;
```

# Flow of control (3)

- **Example with attributes:**

```
s1,s2,s3 : integer:=0;
subtype index is integer range 1..10;
...
for i in 1..10 loop
    s1:=s1+i;
end loop;

for j in index'first..index'last loop
    s2:=s2+j;
end loop;

for k in index'range loop
    s3:=s3+k;
end loop;
```

# Inputs/Outputs (1)

- **Strong Typing:** each type must have its own Inputs/Outputs subprograms, fortunately type families exist.
- **Services provided by package `Text_Io` :** for types `String` and `Character` only:
  - `Get` : read a constant size string from the keyboard.
  - `Put` : display a string on the screen.
  - `New_Line` : send a carriage return to the screen
  - `Put_Line` : `Put` + `New_Line`
  - `Get_Line` : read a variable size string from the keyboard.
- **Other types :** generic units `Float_Io`, `Integer_Io, Enumeration_Io, ...`

# Inputs/Outputs (2)

- **Specification of `Text_Io`:**

```
package Ada.Text_IO is
   procedure Get (Item : out String);
   procedure Put (Item : String);
   procedure Get_Line (Item : out String;
      Last : out Natural);
   procedure Put_Line (Item : String);
   procedure New_Line (Spacing : Positive_Count := 1);

   generic
      type Num is range <>;
   package Integer_IO is ...

   generic
      type Num is range <>;
   package Enumeration_IO is ...
```

# Inputs/Outputs (3)

- **Part of the generic unit `Integer_Io`:**

```
generic
   type Num is range <>;
package Ada.Text_IO.Integer_IO is

   Default_Width : Field := Num'Width;
   Default_Base  : Number_Base := 10;

   procedure Put
     (Item   : Num;
      Width  : Field := Default_Width;
      Base   : Number_Base := Default_Base);
   procedure Get
     (Item : out Num;
      Last : out Positive);
end Ada.Text_IO.Integer_IO;
```

# Inputs/Outputs (4)

- **Example of use of `Integer_Io`:**

```
with text_io; use text_io;
procedure Intio is
   type temperature is new integer range -300..300;
   package temperature_io is new text_io.integer_io(temperature);
   t1, t2 : temperature;
begin
   Put("Get temperature 1:");
   temperature_io.Get(t1);
   New_Line;
   Put("Get temperature 2:");
   temperature_io.Get(t2);
   New_Line;
   Put("Sum = "); temperature_io.Put(t1+t2);
   New_Line;
exception
   when Data_Error =>
     Put_line("Data is not compliant with 'temperature' type");
end Intio;
```

# Inputs/Outputs (5)

- **Exercise 3:**

Write a program that reads integers from the keyboard and displays the sum of the read integers each time an integer is entered. If a read data is not compliant with the integer type, our program must display it.

# Pointers, dynamic allocations (1)

- **Usually, no pointer and dynamic allocations in real-time ... but**
  - **Strong typing:** a pointer can only address a data with the same type. Pointers are typed!
  - **Static analysis on pointer:** reliability.
  - **Example of declarations :**

```
type integer_ptr is access integer;
pointer1 : integer_ptr := null;
my_integer : integer;
pointer2 : integer_ptr := my_integer'access;
```

  - **Dynamic allocation:** new operator.
  - **Deallocation:** none! Why?

# Pointers, dynamic allocations (2)

- **Example:**

```
with Text_Io; use Text_Io;
procedure Pointer is
   package Io is new Text_Io.Integer_Io(Integer);
   type Integer_Ptr is access Integer;
   I   : Integer      := 110;
   P1, P2, P3, P4 : Integer_Ptr;
begin
   P1:= new Integer;
   P1.all:=100;
   P2:= new Integer'(I);
   P4:= new Integer'(10);
   Io.Put(P1.all);
   Io.Put(P2.all);
   Io.Put(P4.all);
   Io.Put(P3.all);
end Pointer;
```

# Pointers, dynamic allocations (3)

- **Static analysis on pointers:** reliabiliy.

```
with Text_Io; use Text_Io;
procedure Wrong_Allocate is

   type Integer_Ptr is access Integer;
   Global : Integer_Ptr;

   procedure Assign_Value is
   I : Integer := 100;
   begin
     Global:=I'access;
   end Assign_Value;

   package Io is new Text_Io.Integer_Io(Integer);
begin
   Assign_Value;
   Io.Put(Global.all);
end Wrong_Allocate;
```

# Generic units (1)

- **Program unit that is parametrized by:** types, constants, subprograms and packages.
- Generic functions/procedures or packages.
- Provide the same service on different types: we can not use object and dynamic linking (those mechanisms are not timely deterministic).
- **Instanciation step:** required to use a generic package. Consists in giving a value for each parameter of the generic unit.
- **Structure:**

```
generic
-- parameters
package foo ...
package body foo ...
-- use parameters to write the
-- implementation of the generic unit
```

# Generic units (2)

```
generic
   type Element is private;
   with procedure Put(E : in Element);
package Lists is
   type Element_Ptr is access Element;
   type Cell is private;
   type Link is access Cell;

   procedure Put(L : in Link);
   procedure Add(L : in out Link;  E : in Element_Ptr);
private
   type Cell is record
        Next     : Link;
        Data     : Element_Ptr;
   end record;
end Lists;
```

# Generic units (3)

```
package body Lists is
   procedure Add(L : in out Link; E : in Element_Ptr) is
      New_Cell : Lien;
   begin
      New_Cell:=new Cell;
      New_Cell.Data:=E; New_Cell.Next:=L; L:=New_Cell;
   end Add;

   procedure Put(L : in Link) is
      Current : Link := L;
   begin
      while Current/=null loop
         Put(Current.Data.all);
         Current:=Current.Next;
      end loop;
   end Put;
end Lists;
```

# Generic units (4)

```
with Lists;
procedure Test_Lists is

   type Guy is record  ...
   procedure Put(Display : in Guy) is  ...

   package My_List is new Lists(Guy, Display);
   use My_List;

   A_List : Link;
   G      : My_List.Element_Ptr;  -- pointer to
                                  -- a guy
begin
   G:= new Guy;
   Add(A_List, G);
   Put(A_List);
   ...
```

# Summary

1. Introduction and sequential programming.
2. Concurrency features.
3. Real-Time features.
4. Examples of Ada runtimes.
5. Conclusion.
6. References.

# Concurrency

- Tasks.
- Synchronization and communication with rendez vous.
- Communication with protected objects.

# Task (1)

- **An Ada task is composed of:**
  - A specification: interface of the task. Visible part of the component.
  - An implementation: contains the source code of the task (statements sequentially run by the task). Hidden part.
  - Optional type (anonymous task otherwise).
- **An Ada task is declared as follow:**
  - `task/task type` (specification of the task) and `task body` (implementation of the task).
  - A main procedure is also a task.

# Task (2)

- **A task can be:** active, aborted, achieved, terminated.
- **Activation rules:**
  - Statically allocated: in the beginning of the bloc in which the task is declared.
  - Dynamically allocated: at dynamic allocation (new statement).
- **Termination rules:**
  - On exception (exceptions in a task are lost if not catched).
  - When all slaves tasks are terminated.
- **Abortion :** with `abort x` statement, `x` is a task name. Should be avoided.

# Task (3)

- **Example of an anonymous task, statically allocated:**

```
with Text_Io; use Text_Io;
procedure Anonymous_Task is
   task My_Task;
   task body My_Task is
   begin
      loop
         Put_Line("Task is running");
         delay 1.0;
      end loop;
   end My_Task;
begin
   null;
end Anonymous_Task;
```

- How many tasks here?

# Task (4)

- **Example of a task type, statically allocated:**

```
with Text_Io; use Text_Io;
procedure Task_Type is
   task type A_Type;
   task body A_Type is
   begin
      loop
         Put_Line("task is running");
         delay 1.0;
      end loop;
   end A_Type;
   T1, T2: A_Type;
   T : array (1..10) of A_Type;
begin
   null;
end Task_Type;
```

- How many tasks here?

# Task (5)

- **Example of a task type, dynamically allocated:**

```
with Text_Io; use Text_Io;
procedure Dynamic_Task is
   task type A_Type;
   task body A_Type is
   begin
      loop
         Put_Line("task is running");
         delay 1.0;
      end loop;
   end A_Type;
   type A_Type_Ptr is access A_Type;
   T : array (1 .. 3) of A_Type_Ptr;
begin
   for i in 1..3 loop
      T(i):= new A_Type;
   end loop;
end Dynamic_Task;
```

# Task (6)

- This program is wrong. Why?

```
procedure Wrong_Task is
   cpt : integer :=0;
   task type A_Type;
   task body A_Type is
   begin
      loop
         cpt:=cpt+1;
         delay 1.0;
      end loop;
   end A_Type;
   T1, T2 : A_Type;
begin
   delay 3.0;
   cpt:=cpt+1;
   abort T1; abort T2;
end Wrong_Task;
```

# Task (7)

- **Exercise 4:**

For programs of pages 46, 47 and 48, say when the tasks are activated and when they are terminated.

# Task (8)

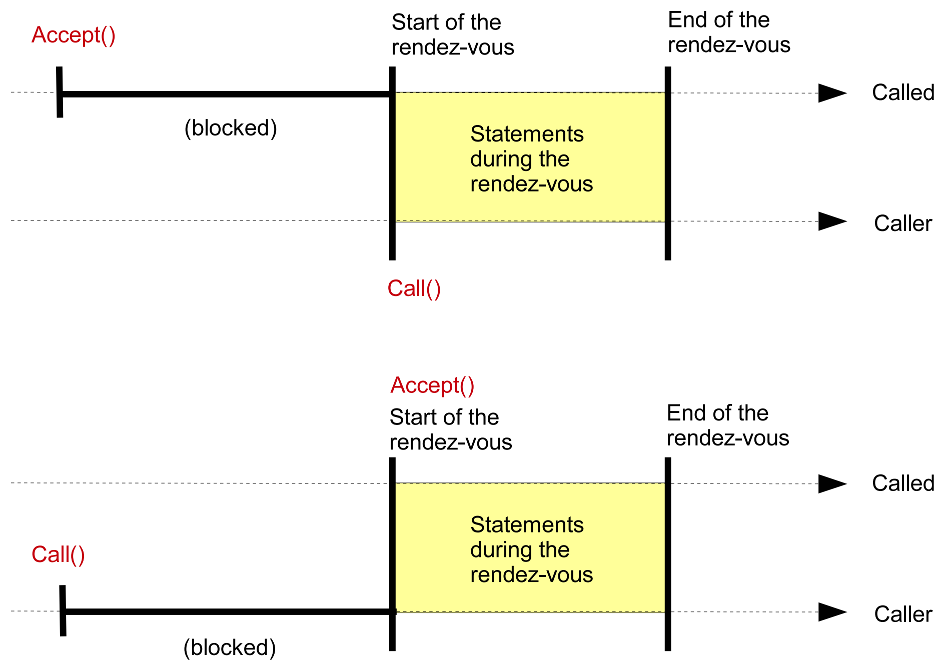
- **Exercise 5:**

Write a program composed of two tasks. The first task computes and displays the elements of the recurrent equation $U_n = U_{n-1} \cdot 2$ with $U_0 = 1$. The task must be delayed one second between the display of two successive values. The second task must have the same behavior but with the following recurrent equation: $U_n = U_{n-1} + 2$ with $U_0 = 0$.

# Concurrency

- Tasks.
- Synchronization and communication with rendez vous.
- Communication with protected objects.

# Rendez-vous (1)




- **A rendez-vous is:**
  - **Asymmetric:** calling task and called task. Server and client.
  - **Synchronization:** two tasks must be ready to do the rendez-vous.
  - Allow data exchange between two tasks.

# Rendez-vous (2)

- **What is a rendez-vous with Ada:**
  - **Entry:** synchronization point, declared in the task specification (task interface).
  - **Task specification:** may include several entries.
  - **`accept` statement:** allow a called task to wait for an entry call, and then, during the rendez-vous, runs statements included in the accept.
  - **Caller task:** callers use entry names to make a rendez-vous with a task.

# Rendez-vous (3)

- **Example of rendez-vous (synchronization only):**

```
with Text_Io; use Text_Io;
procedure Hello is
   task My_Task is
      entry Hello_World;
   end My_Task;
   task body My_Task is
   begin
      loop
         accept Hello_World do
            Put_Line("Hello word");
         end Hello_World;
      end loop;
   end My_Task;
begin
   My_Task.Hello_World;
   abort My_Task;
end Hello;
```

# Rendez-vous (4)

- **Rendez-vous with data exchange:**

```
  task type My_Task is
     entry Increment(S1 : in out Integer);
  end My_Task;
  task body My_Task is
  begin
     loop
           accept Increment(S1 : in out Integer) do
              S1:=S1+1;
           end Increment;
     end loop;
  end My_Task;
  T1 : My_Task;
  Val : Integer :=0;
begin
  T1.Increment(Val);
  Put_Line("Val = " & Integer'Image(Val));
  abort T1;
```

# Rendez-vous (5)

- **Some additional clauses:**
  - **select :** simultaneously wait for several entry calls.
  - **terminate :** stop a task which is waiting for an entry call when its master task is completed.
  - **Guarded entry:** the entry can be called only when the boolean condition is satisfied.
  - **else :** non blocking entry call.
  - ...

# Rendez-vous (6)

- **Task with several entries:**

```
task body My_Task is
   Bool : Boolean := False;
begin
   loop
      select
         accept Hello_World do
            Put_Line("Hello word");
         end Hello_World;
      or
         accept Do_Exit do
            Put_Line("Bye bye");
            Bool:=True;
         end Do_Exit;
      end select;
      exit when Bool;
   end loop;
end My_Task;
```

```
task My_Task is
   entry Hello_World;
   entry Do_Exit;
end My_Task;
```

```
begin
   My_Task.Hello_World;
   My_Task.Do_Exit;
end Main;
```

# Rendez-vous (7)

- **Example with a `terminate` clause:**

```
task body My_Task is
begin
   loop
      select
         accept Increment
           (S1: in out Integer) do
            S1:=S1+1;
         end Increment;
      or
         terminate;
      end select;
   end loop;
end My_Task;
```

```
task My_Task is
   entry Increment
      (S1 : in out Integer);
end My_Task;

Val : Integer :=0;

begin
   My_Task.Increment(Val);
   Put_line("Val = " &
         Integer'Image(Val));
end Increment_Terminate;
```

# Concurrency

- Tasks.
- Synchronization and communication with rendez-vous.
- Communication with protected objects.

# Protected types and objects (1)

- **A protected object is:**
  - A structure protecting concurrent access to a set of variables.
  - A synchronization mechanism which is a kind of readers-writers paradigm.
- **A protected object is composed of:**
  - A specification (interface): declaration of functions, procedures and entries. Visible part of the unit.
  - An implementation (body): protected variables + implementation of the functions, procedures and entries. Hidden part of the unit.
  - Optional type (anonymous protected object otherwise).

# Protected types and objects (2)

- **Synchronizations provided by a protected object:**
  - **Functions:** can be run simultaneously (concurrency is allowed on functions as they do not change object data).
  - **Procedures:** concurrency is not allowed: source code of a procedure is run in a critical section.

    $\Longrightarrow$ Function and procedure synchronization = readers-writers synchronization.
  - **Entry:** similar to procedures but with a boolean guard: an entry can be run only if its guard is true, blocking of the task otherwise.

# Protected types and objects (3)

- **Example of a protected variable (specification):**

```
package Vars is
   protected type Var is
      procedure Write(Value : in Integer);
      function Read return Integer;
   private
      Variable : Integer:=0;
   end Var;
end Vars;
```

# Protected types and objects (4)

- **Example of a protected variable (body):**

```
package body Vars is
   protected body Var is
      procedure Write(Value : in Integer) is
      begin
         Variable:=Value;
      end Write;
      function Read return Integer is
      begin
         return Variable;
      end Read;
   end Var;
end Vars;
```

# Protected types and objects (5)

- **Example of a protected variable (use):**

```
With Text_Io; use Text_io;
with Vars; use Vars;
procedure Protected_Variable is
   One : Vars.Var;
   task My_Task;
   task body My_Task is
   begin
      loop
         Put_Line("Val = " & Integer'Image(One.Read));
      end loop;
   end My_Task;
   I : Integer :=0;
begin
   loop
      One.Write(I);
      I:=I+1;
   end loop;
end Protected_Variable;
```

# Protected types and objects (6)

- **Example of a semaphore (specification):**

```
package Semaphores is
   protected type Semaphore is
      entry P;
      procedure V;
      procedure Init (
            Val : in Natural );
   private
      Value : Natural:=1;
   end Semaphore;
end Semaphores;
```

# Protected types and objects (7)

- **Example of a semaphore (body):**

```
package body Semaphores is
   protected body Semaphore is
      entry P when Value > 0 is
      begin
         Value:=Value-1;
      end P;
      procedure V is
      begin
         Value:=Value+1;
      end V;
      procedure Init ( Val : in Natural ) is
      begin
         Value:=Val;
      end Init;
   end Semaphore;
end Semaphores;
```

# Protected types and objects (8)

```
Mutex : Semaphore;
task type One;
task body One is
begin
   loop
      Mutex.P;
      Put_Line("Running in critical section !!");
      Mutex.V;
   end loop;
end One;

type One_Ptr is access One;
Several : array (1..10) of One_Ptr;

begin
   Mutex.Init(1);
   for i in 1..10 loop
        Several(i):= new One;
```

# Protected types and objects (9)

- **Example of a readers-writers synchronization:**

```
package Readers_Writers is
   protected type Reader_Writer is
      entry Start_Read;
      procedure End_Read;
      entry Start_Write;
      procedure End_Write;
   private
      Nb_Readers : Natural :=0;
      Nb_Writers : Natural :=0;
   end Reader_Writer;
end Readers_Writers;
```

# Protected types and objects (10)

```
protected body Reader_Writer is
    entry Start_Read when Nb_Writers = 0 is
    begin
        Nb_Readers:=Nb_Readers+1;
    end Start_Read;
    entry Start_Write when Nb_Readers + Nb_Writers = 0 is
    begin
        Nb_Writers=Nb_Writers+1;
    end Start_Write;
    procedure End_Read is
    begin
        Nb_Readers:=Nb_Readers−1;
    end End_Read;
    procedure End_Write is
    begin
        Nb_Writers=Nb_Writers−1;
    end End_Write;
end Reader_Writer;
```

# Summary

1. Introduction and sequential programming.
2. Concurrency features.
3. Real-Time features.
4. Examples of Ada runtimes.
5. Conclusion.
6. References.

# Real-time

- **Real-time scheduling facilities available for Ada practitioners:**
  - ISO/IEC Ada 1995 and 2005 : the Systems Programming Annex C and the Real-Time Annex D [TAF 06].
  - Ada POSIX 1003 Binding [BUR 07, GAL 95].
  - ARINC 653 [ARI 97].
  - ...

# Ada 2005 real-time scheduling facilities

- **With Ada 1995/2005, real-time scheduling features are provided by pragmas and specific packages:**

  - How to implement a periodic task:
    1. Representing time ($Ada.Real\_Time$ package).
    2. Implementing periodic release times ($delay$ statement).
    3. Assigning priorities (pragma).
  - How to activate priority inheritance with shared resources (protected objects/types).
  - How to select a scheduler (fixed priority scheduling, EDF, ...).
  - ...

# Ada 2005: periodic task (1)

```
package Ada.Real_Time is

   type Time is private;
   Time_Unit  : constant := implementation-defined;
   type Time_Span is private;
   ...
   function Clock return Time;
   ...
   function Nanoseconds  (NS : Integer) return Time_Span;
   function Microseconds (US : Integer) return Time_Span;
   function Milliseconds (MS : Integer) return Time_Span;
   function Seconds (S : Integer) return Time_Span;
   function Minutes (M : Integer) return Time_Span;
   ...
```

- $Ada.Real\_Time$ provides a new **monotonic**, **high-resolution** and **documented** "Calendar" package.

# Ada 2005: periodic task (2)

- $Time$ implements an absolute time. The range of this type shall be sufficient to represent real ranges up to 50 years later.
- $Time\_Span$ represents the length of real-time duration.
- $Time\_Unit$ is the smallest amount of real-time representable by the $Time$ type. It is implementation defined. Shall be less than or equal to 20 microseconds.
- $Clock$ returns the amount of time since $epoch$.
- Some sub-programs which convert input parameters to $Time\_Span$ values (e.g. $Nanoseconds$, $Microseconds$, ...).

# Ada 2005: periodic task (3)

- **Implementing periodic release times with $delay$ statements:**

1. $delay\ expr$ : blocks a task during **at least** $expr$ amount of time.
2. $delay\ until\ expr$ : blocks a task until **at least** the absolute time expressed by $expr$ is reached.

- A task can not be released **before the amount of time** specified with the $delay$ statement.
- But tasks can be released **after the amount of time** specified with the $delay$ statement
- No upper bound on the release time lateness for a $delay$ statement.
- Upper bound lateness shall be documented by the implementation.

# Ada 2005: periodic task (4)

- **Example of a periodic task (car embedded software example):**

```
with Ada.Real_Time; use Ada.Real_Time;
 ...
 task Tspeed is
 end Tspeed;

 task body Tspeed is
    Next_Time : Ada.Real_Time.Time := Clock;
    Period : constant Time_Span := Milliseconds (250);
 begin
    loop
       --  Read the car speed sensor
       ...
       Next_Time := Next_Time + Period;
       delay until Next_Time;
    end loop;
 end Tspeed;
```

- Use $delay\ until$ instead of $delay$ (due to clock cumulative drift).

# Ada 2005: periodic task (5)

- **Ada priority model :**

```
package System is

   --  Priority-related Declarations (RM D.1)
   Max_Priority           : constant Positive := 30;
   Max_Interrupt_Priority : constant Positive := 31;

   subtype Any_Priority       is Integer      range  0 .. 31;
   subtype Priority           is Any_Priority range  0 .. 30;
   subtype Interrupt_Priority is Any_Priority range 31 .. 31;

   Default_Priority : constant Priority := 15;
   ...
```

- **Base** priority : statically assigned.
- **Active** priority : inherited (rendez-vous, ICPP/protected objects).
- $System.Priority$ must provide at least 30 priority levels (but having more levels is better for real-time scheduling analysis).

# Ada 2005: periodic task (6)

- **Task base priority assignment rules with Ada 1995/2005:**
  - Priority pragma can be used in task specifications.
  - Priority pragma can be assigned to main procedures.
  - Any task without Priority pragma has a priority equal to the task that created it.
  - Any task has a default priority value (see the $System$ package).

# Ada 2005: periodic task (7)

- **Declaring a task:**

```
task Tspeed is
   pragma Priority (10);
end Tspeed;
```

- **Declaring with a task type:**

```
task type T is
   pragma Priority (10);
end T;
Tspeed : T
```

- **Declaring with a task type and a discriminant:**

```
task type T (My_Priority : System.Priority) is
   entry Service( ...
   pragma Priority (My_Priority);
end T;
Tspeed : T(My_Priority =>10);
```

# Ada 2005: periodic task (8)

- **Let assume this task set:**

| Task | Period (milli-secondes) | Priority |
|---|---|---|
| $T_{display}$ | $P_{display} = 100$ | 12 |
| $T_{engine}$ | $P_{engine} = 500$ | 10 |
| $T_{speed}$ | $P_{speed} = 250$ | 11 |

- **And their source code:**

```
procedure Display_Speed is
begin
   Put_Line ("Tdisplay displays the speed of the car");
end Display_Speed;

procedure Read_Speed is ...
procedure Monitor_Engine is ...
```

# Ada 2005: periodic task (9)

```
with System;
generic
   with procedure Run;
package Generic_Periodic_Task is
   task type Periodic_Task (Task_Priority : System.Priority;
                            Period_In_Milliseconds : Natural) is
      pragma Priority (Task_Priority);
   end Periodic_Task;
end Generic_Periodic_Task;
```

# Ada 2005: periodic task (10)

```
package body Generic_Periodic_Task is
   task body Periodic_Task is
      Next_Time : Ada.Real_Time.Time := Clock;
      Period    : constant Time_Span :=
                  Milliseconds (Period_In_Milliseconds);
   begin
      loop
         Run;
         Next_Time := Next_Time + Period;
         delay until Next_Time;
      end loop;
   end Periodic_Task;
end Generic_Periodic_Task;
```

# Ada 2005: periodic task (11)

```
procedure Car_System is
   package P1 is new Generic_Periodic_Task (Run => Display_Speed);
   package P2 is new Generic_Periodic_Task (Run => Read_Speed);
   package P3 is new Generic_Periodic_Task (Run => Monitor_Engine);

   Tdisplay : P1.Periodic_Task (Task_Priority => 12,
                                Period_In_Milliseconds => 100);
   Tspeed   : P2.Periodic_Task (Task_Priority => 11,
                                Period_In_Milliseconds => 250);
   Tengine  : P3.Periodic_Task (Task_Priority => 10,
                                Period_In_Milliseconds => 500);

   pragma Priority (20);

begin
   Put_Line ("All tasks start when the main procedure completes");
end Car_System;
```

# Ada 2005: protected objects (1)

- **Inheritance priority protocols supposed by Ada 2005:** ICPP (Immediate Ceiling Priority Protocol) and PLCP (Preemption Level Control Protocol).

- **ICPP is a kind of PCP that works as follows:**
  - Ceiling priority of a resource = maximum static priority of the tasks which use it.
  - Dynamic task priority = maximum of its own static priority and the ceiling priorities of any resources it has locked.

# Ada 2005: protected objects (2)

- **Assignment of a ceiling priority to a protected object:**

```
protected A_Mutex is
   pragma Priority(15);
   entry E ...
   procedure P...
end A_Mutex;
```

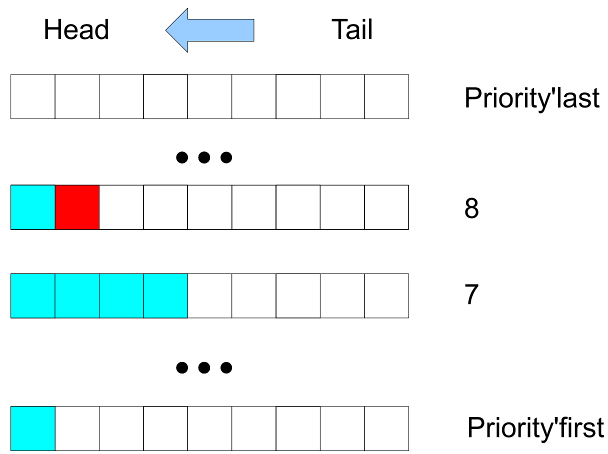
- **To activate ICPP on protected objects:**

```
pragma Locking_Policy(Ceiling_Locking);
```

# Ada 2005 real-time scheduling facilities

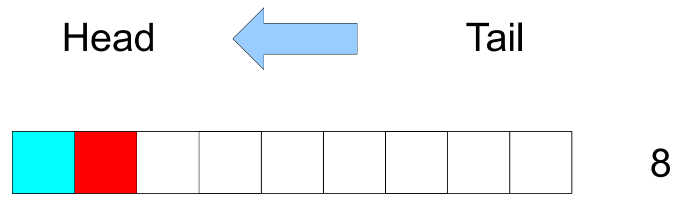
- **With Ada 1995/2005, real-time scheduling features are provided by pragmas and specific packages:**
  - How to implement a periodic task:
    1. Representing time ($Ada.Real\_Time$ package).
    2. Implementing periodic release times ($delay$ statement).
    3. Assigning priorities (pragma).
  - How to activate priority inheritance with shared resources (protected objects/types).
  - How to select a scheduler (fixed priority scheduling, EDF, ...).
  - ...

# Ada 2005 scheduling model (1)




- **Ada 2005 real-time scheduling model:**
  - A queue for each priority level. All ready tasks which have the same active priority level are put in the same queue.
  - Each queue has a dispatching policy.
  - Two-levels of scheduling:
    1. Choose the highest priority queue with at least one ready task.
    2. Choose the task to run of the queue selected in (1), according to the queue dispatching policy.

# Ada 2005 scheduling model (2)




- Example of the preemptive $FIFO\_Within\_Priorities$ dispatching policy:
  - When a task becomes ready, it is inserted in the tail of its corresponding priority queue.
  - The task at the head of the queue gets the processor when it becomes the highest ready priority task/queue.
  - When a running task becomes blocked or terminated, it leaves the queue and the next task in the queue gets the processor.

$\Longrightarrow$ **We can easily apply fixed priority scheduling feasibility tests if all tasks have different priority levels.**

# Ada 2005 scheduling model (3)

- **The** $FIFO\_Within\_Priorities$ **dispatching policy is activated by:**

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

- **Ada 2005 also provides other dispatching policies:**

1. Non preemptive fixed priority dispatching:

```
pragma Task_Dispatching_Policy(
   Non_Preemptive_FIFO_Within_Priorities);
```

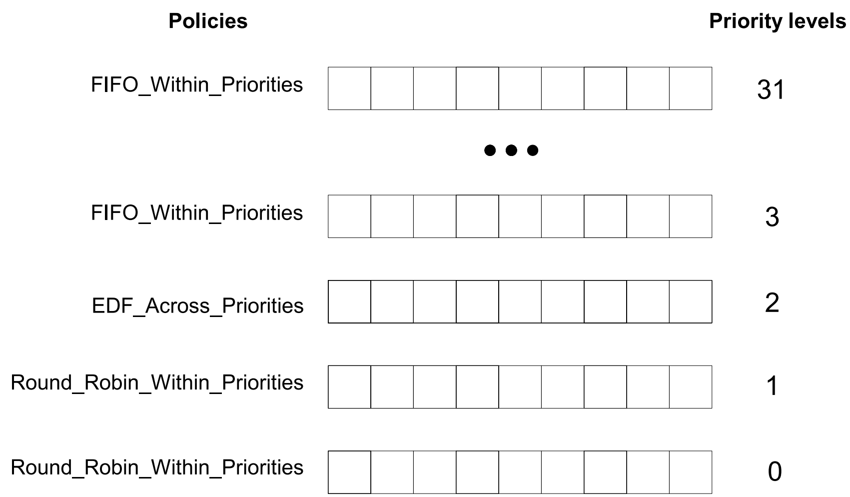
2. Earliest deadline first dispatching:

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

3. Round robin dispatching:

```
pragma Task_Dispatching_Policy(
   Round_Robin_Within_Priorities);
```

# Ada 2005 scheduling model (4)




- **We can run altogether critical and non critical tasks by mixing dispatching protocols.** Each priority level may have its own dispatching protocol:

```
pragma Priority_Specific_Dispatching(
    FIFO_Within_Priorities, 3, 31);
pragma Priority_Specific_Dispatching(
    EDF_Across_Priorities, 2, 2);
pragma Priority_Specific_Dispatching(
    Round_Robin_Within_Priorities, 0, 1);
```

# Ada 2005 scheduling model (5)

- **Example of the software embedded into a car:**

```
procedure Car_System is
...
   Tdisplay  : P1.Periodic_Task (Task_Priority => 12,
                                 Period_In_Milliseconds => 100);
   Tspeed    : P2.Periodic_Task (Task_Priority => 11,
                                 Period_In_Milliseconds => 250);
   Tengine   : P3.Periodic_Task (Task_Priority => 10,
                                 Period_In_Milliseconds => 500);
    pragma Priority (20);
...
end Car_System;

-- File gnat.adc  (or directly in the compilation unit)
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
```

# Ada 2005 Ravenscar profile (1)

- **Remember the feasibility tests examples:** processor utilization factor test, worst case response time.
- **Each feasibility test has several applicability assumptions.** Processor utilization factor test assumes:
  - Fixed preemptive scheduling.
  - Rate monotonic priority assignment.
  - ICCP shared resources/protected object.
  - Periodic release times.
  - Critical instant.
  - ...
- **How to be sure that your applications is compliant with those feasibility tests assumptions ?**
- **How to increase compliance of your applications with feasibility tests ?** $\Longrightarrow$ use Ravenscar.

# Ada 2005 Ravenscar profile (2)

- **What is Ravenscar:**
  - Ravenscar defines an Ada sub-language which is compliant with Rate Monotonic feasibility tests.
  - Ravenscar is a profile which is part of the Ada 2005 standard.
  - A profile is a set of restrictions a program must meet.
  - Restrictions are expressed with pragmas. They are checked at compile-time to enforce the restrictions at execution time.

# Ada 2005 Ravenscar profile (3)

- The Ravenscar profile is activated by:

```
pragma profile(Ravenscar);
```

- Examples of the restrictions enforced by Ravenscar:

```
-- Use preemptive fixed priority scheduling
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);

--  Use ICPP
pragma Locking_Policy(Ceiling_Locking);

pragma Restrictions(
  No_Task_Allocators,   -- No task dynamic allocation
                        -- ASSUMPTION RELATED TO  TASK
                        -- THE CRITICAL INSTANT
  No_Dependence => Ada.Calendar,   -- Use Real-time calendar only
  No_Relative_Delay,   -- Disallow time drifting due to
                       -- the use of the delay statement
  ...
  );
```

# Real-time

- **Real-time scheduling facilities available for Ada practitioners:**
  - ISO/IEC Ada 1995 and 2005 : the Systems Programming Annex C and the Real-Time Annex D [TAF 06].
  - Ada POSIX 1003 Binding [BUR 07, GAL 95].
  - ARINC 653 [ARI 97].
  - ...

# POSIX 1003 standard (1)

- Define a standardized interface of an operating system similar to UNIX [VAH 96].

- Published by ISO and IEEE. Organized in chapters:

| Chapters | Meaning |
|---|---|
| POSIX 1003.1 | System Application Program Interface (e.g. $fork$, $exec$) |
| POSIX 1003.2 | Shell and utilities (e.g. $sh$) |
| POSIX 1003.1b [GAL 95] | Real-time extensions. |
| POSIX 1003.1c [GAL 95] | Threads |
| POSIX 1003.5 | Ada POSIX binding |
| ... | |

- Each chapter provides a set of services. A service may be mandatory or optional.

# POSIX 1003 standard (2)

- Example of operating systems providing 1003.1b : Lynx/OS, VxWorks, Solaris, Linux, QNX, etc .. (actually, most of real-time operating systems).
- POSIX 1003.1b services :

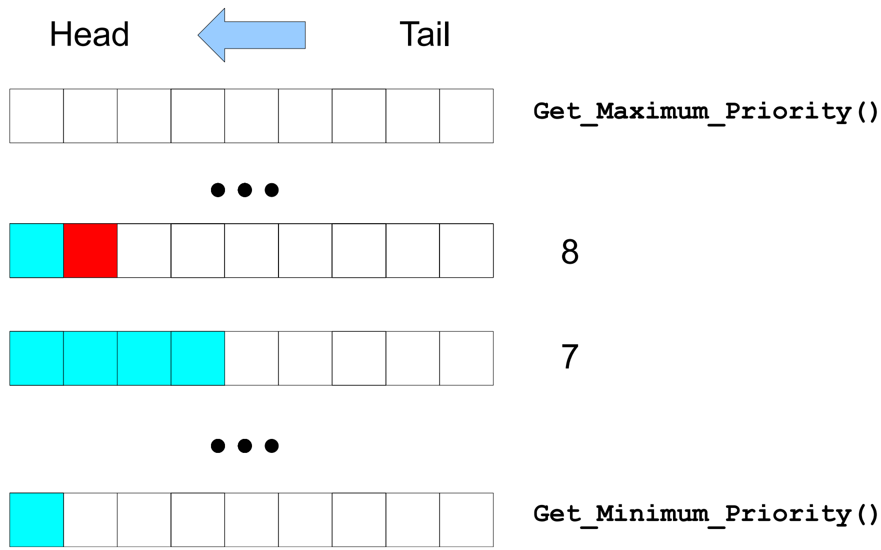
| Name | Meaning |
|---|---|
| _POSIX_PRIORITY_SCHEDULING | Fixed priority scheduling |
| _POSIX_REALTIME_SIGNALS | Real-time signals |
| _POSIX_ASYNCHRONOUS_IO | Asynchronous I/O |
| _POSIX_TIMERS | WatchDogs |
| _POSIX_SEMAPHORES | Synchronization tools |
| ... | |

# POSIX 1003 standard (3)

- How the Ada programmer can run POSIX 1003.1b applications ? POSIX 1003.5 Ada binding (e.g. Florist).

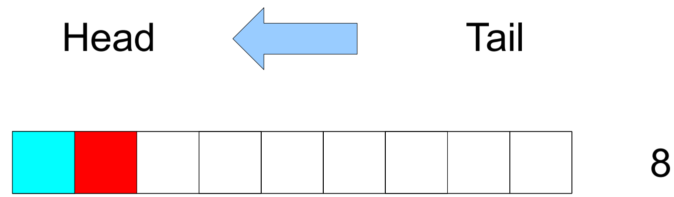
- **This Ada binding provides access to POSIX 1003:**
  - Scheduling services for fixed priority scheduling, EDF, ...
  - Timers to implement periodic release times.

# POSIX 1003 standard (4)




- **POSIX real-time scheduling model:**
  - Preemptive fixed priority scheduling. At least 32 priority levels.
  - Two-levels scheduling :
    1. Choose the queue which has the highest priority level with at least one ready process/thread.
    2. Choose a process/thread from the queue selected in (1) according to a **policy**.

# POSIX 1003 standard (5)




- **POSIX policies:**

1. $SCHED\_FIFO$ : similar to the $FIFO\_Within\_Priorities$. Ready processes/threads of a given priority level get the processor according to their order in the queue.
2. $SCHED\_RR$ : $SCHED\_FIFO$ with a time quantum. A time quantum is a maximum duration that a process/thread can run on the processor before preemption by an other process/thread of the same queue. When the quantum is exhausted, the preempted process/thread is moved to the tail of the queue.
3. $SCHED\_OTHER$ : implementation defined (usually implements a time sharing scheduler).

# POSIX 1003 standard (6)

- **Example of the** $Process\_Scheduling$ **package which defines:**
  - Priority/policy types.
  - Sub-programs to adapt POSIX application to RTOS features.
  - Sub-programs to change scheduling properties of processes.

```
package POSIX.Process_Scheduling is

   subtype Scheduling_Priority is Integer;

   type Scheduling_Policy is new Integer;
   Sched_FIFO  : constant Scheduling_Policy := ...
   Sched_RR    : constant Scheduling_Policy := ...
   Sched_Other : constant Scheduling_Policy := ...

   type Scheduling_Parameters is private;
```

# POSIX 1003 standard (7)

- **Sub-programs which allow the application to adapt itself to the underlying real-time operating system:**

```
package POSIX.Process_Scheduling is
   ...
 function Get_Maximum_Priority (Policy:Scheduling_Policy)
     return Scheduling_Priority;
 function Get_Minimum_Priority (Policy:Scheduling_Policy)
     return Scheduling_Priority;

 function Get_Round_Robin_Interval
     (Process : POSIX_Process_Identification.Process_ID)
     return POSIX.Timespec;
   ...
```

# POSIX 1003 standard (8)

- **Set or get policy/priority of a process:**

```
package POSIX.Process_Scheduling is

   procedure Set_Priority
     (Parameters : in out Scheduling_Parameters;
      Priority   : Scheduling_Priority);
   procedure Set_Scheduling_Policy
     (Process    : POSIX_Process_Identification.Process_ID;
      New_Policy : Scheduling_Policy;
      Parameters : Scheduling_Parameters);
   procedure Set_Scheduling_Parameters
     (Process    : POSIX_Process_Identification.Process_ID;
      Parameters : Scheduling_Parameters);

   function Get_Scheduling_Policy ...
   function Get_Priority ...
   function Get_Scheduling_Parameters ...
```

# POSIX 1003 standard (9)

- **Example of the car embedded software example:**

```
with POSIX.Process_Identification; use POSIX.Process_Identification;
with POSIX.Process_Scheduling; use POSIX.Process_Scheduling;

   Pid1 : Process_ID;
   Sched1 : Scheduling_Parameters;

begin
   Pid1:=Get_Process_Id;

   Sched1:=Get_Scheduling_Parameters(Pid1);
   Put_Line("Current priority/policy = "
            & Integer'Image(Get_Priority(Sched1))
            & Integer'Image(Get_Scheduling_Policy(Pid1)));

   Set_Priority(Sched1, 10);
   Set_Scheduling_Policy(Pid1, SCHED_FIFO, Sched1);
   Set_Scheduling_Parameters(Pid1, Sched1);
```

# POSIX 1003 standard (10)

- **Does an Ada programmer should use POSIX Ada binding ?**
- **Nice sides of POSIX:**
  - POSIX is supported by a large number of RTOS.
  - Analysis with feasibility tests can be performed with the POSIX scheduling framework.
- **But POSIX also has some drawbacks:**
  - What is a POSIX process ? a POSIX thread ? a task ?
  - Programs may be more complex (timers to implement periodic task releases, use of scheduling services).
  - No Ravenscar to handle feasibility test assumptions.
  - Does POSIX really portable since many services are optional ?

# Summary

1. Introduction and sequential programming.
2. Concurrency features.
3. Real-Time features.
4. Examples of Ada runtimes.
5. Conclusion.
6. References.

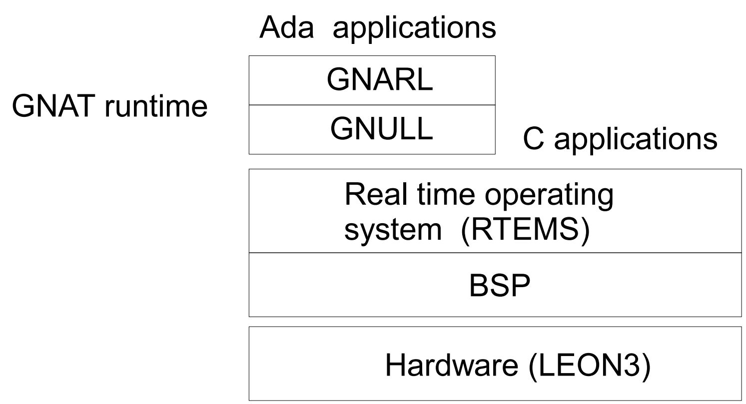
# Examples of Ada runtimes (1)

- **What is a runtime:**
  - Library providing execution environment for Ada programs.
  - Adapt the operating system services to the one required for Ada features: tasks, protected object, priority.
  - Warning: a runtime may not provide all Ada features:
    1. Compiler may help to detect missing features.
    2. Package `System` describes available services.

# Examples of Ada runtimes (2)

- The Open-Ravenscar project, ORK operating system with Ada 2005 scheduling and POSIX binding. (Universidad Politécnica de Madrid, `http://polaris.dit.upm.es/~ork/`).
- Marte operating system, implemented with AdaCore GNAT compiler. (Universidad de Cantabria, `http://marte.unican.es/`)
- GNAT GPL, Ada 2005 scheduling and POSIX binding (Florist). GNAT Runtime available for Windows, Linux, Solaris and numerous real time operating systems. (AdaCore, `http://www.adacore.com/`).
- RTEMS operating system (OAR Corporation, `http://www.rtems.com/`).
- ...

# Examples of Ada runtimes (3)




- **RTEMS Runtime:**
  - RTEMS : real-time operating systems for high critical application. Low memory footprint.
  - Available for many BSP (including Leon processor: 32 bits, VHDL open-source, SMP ou AMP, compatible SPARC, devoted for space/aircraft applications).
  - GNAT compiler (AdaCore company).
  - Cross-compiling on Linux, Leon will be our target system.

# Examples of Ada runtimes (4)

- **Cross-compiling:**
  1. Compile on Linux and generate SPARC binaries:

```
#sparc-rtems4.8-gnatmake hello
sparc-rtems4.8-gcc -c hello.adb
sparc-rtems4.8-gnatbind hello.ali
sparc-rtems4.8-gnatlink hello.ali -o hello.obj
sparc-rtems4.8-size hello.obj
   text    data     bss     dec     hex filename
 288800   13012   17824  319636   4e094 hello.obj
sparc-rtems4.8-nm hello.obj >hello.num
#file hello.obj
hello.obj: ELF 32-bit MSB executable, SPARC, version 1 (SYSV),
statically linked, not stripped
#file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked(uses shared libs),for GNU/Linux 2.6.15,stripped
```

# Examples of Ada runtimes (5)

- **Cross-compiling (cont) :**

  2 Send the binary to a Leon board (serial link, TCP/IP, ...)

  3 Run the program on the Leon board or with an emulator such as `tsim` (Leon processor emulator).

```
#tsim hello.obj
 TSIM/LEON3 SPARC simulator, version 2.0.15 (evaluation version)
allocated 4096 K RAM memory, in 1 bank(s)
allocated 32 M SDRAM memory, in 1 bank
allocated 2048 K ROM memory
read 2257 symbols
tsim> go
resuming at 0x40000000
** Init start **
** Init end **
Hello World
Program exited normally.
tsim>
```

# Examples of Ada runtimes (6)

- **`Systems.ads` package of RTEMS runtime:**

```
package System is
   Tick                        : constant := 0.01;

   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order := High_Order_First;

  -- Priority-related Declarations (RM D.1)
  -- RTEMS provides 0..255 priority levels
  --
   Max_Priority            : constant Positive := 30;
   Max_Interrupt_Priority  : constant Positive := 31;
   subtype Any_Priority       is Integer      range  0 .. 31;
   subtype Priority           is Any_Priority range  0 .. 30;
   subtype Interrupt_Priority is Any_Priority range 31 .. 31;
   Default_Priority : constant Priority := 15;
   ...
```

# Examples of Ada runtimes (7)

- **Runtime GNAT Intel/Linux :**
  - Linux/Intel : non real-time system, but can be used for soft real-time application with POSIX 1003.
  - GNAT compiler.
  - Compliant with Ada 2005 and also POSIX 1003 (Ada/POSIX 1003.5 florist binding)
  - No cross-compiling.

# Examples of Ada runtimes (8)

- **Runtime GNAT Intel/Linux scheduling services:**
  - Compliant with POSIX 1003.
  - Priority 0 for `SCHED_OTHER` (time sharing processes).
  - Priority 1 to 99 for `SCHED_FIFO/SCHED_RR` (real-time processes).
  - Require root privileges to use priority 1 to 99.

  - GNAT Intel/Linux maps Ada priority to Linux priority as follows:
    1. `SCHED_OTHER` processes: Ada priorities are ignored.
    2. `SCHED_RR` or `SCHED_FIFO` processes: direct mapping of Ada task priorities.

# Examples of Ada runtimes (9)

- **`Systems.ads` package of Linux runtime:**

```
package System is
   Tick                        : constant := 0.000_001;

   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order := Low_Order_First;

   -- Priority-related Declarations (RM D.1)
   -- Linux provides 0..99 priority levels (0 for SCHED_OTHER, 1_99
   -- for SCHED_FIFO/SCHED_RR
   --
   Max_Priority            : constant Positive := 97;
   Max_Interrupt_Priority  : constant Positive := 98;
   subtype Any_Priority       is Integer      range  0 .. 98;
   subtype Priority           is Any_Priority range  0 .. 97;
   subtype Interrupt_Priority is Any_Priority range 98 .. 98;
   Default_Priority : constant Priority := 48;
   ...
```

# Summary

1. Introduction and sequential programming.
2. Concurrency features.
3. Real-Time features.
4. Examples of Ada runtimes.
5. Conclusion.
6. References.

# Conclusion

- **Reliable programming:** strong typing and various verifications (access pointer, index, in/out argument), ...
- Separate compilation and large projects.
- Portability.
- **Concurrency, synchronization and communication:** Ada task, rendez-vous, protected objects.
- **Real-time features:** periodic task, priority, EDF/Fixed priority scheduling, ICPP.
- **Ravenscar:** beeing compliant with real-time scheduling analysis.
- **Cross-compiling and runtimes.**

# Summary

1. Introduction and sequential programming.
2. Concurrency features.
3. Real-Time features.
4. Examples of Ada runtimes.
5. Conclusion.
6. References.

# References

[ARI 97] Arinc. *Avionics Application Software Standard Interface*. The Arinc Committee, January 1997.

[BUR 07] A. Burns and A. Wellings. *Concurrent and Real Time programming in Ada. 2007*. Cambridge University Press, 2007.

[GAL 95] B. O. Gallmeister. *POSIX 4 : Programming for the Real World* . O'Reilly and Associates, January 1995.

[LES 10] D. Lesens. « Using Static Analysis in Space. Why doing so ? ». pages 51–70. Proceedings of the SAS 2010 conference, Springer Verlag, LNCS, volume 6337, September 2010.

[TAF 06] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1*. LNCS Springer Verlag, number XXII, volume 4348., 2006.

[VAH 96] U. Vahalia. *UNIX Internals : the new frontiers*. Prentice Hall, 1996.