

---

# Développement de systèmes embarqués temps réel avec Ada

Frank Singhoff

Bureau C-207

Université de Brest, France

LISyC/EA 3883

singhoff@univ-brest.fr

[MCours.com](http://MCours.com)

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.

# Présentation

---

- **Caractéristiques des systèmes embarqués temps réel et objectifs :**

1. Comme tous systèmes temps réel : déterminisme logique, temporel et fiabilité.

2. Mais en plus :

- Ressources limitées (mémoire <sup>a</sup>, vitesse processeur, énergie).
- Accessibilité réduite.
- Autonomie élevée.
- Interaction avec son environnement (capteurs).

⇒ **Environnements d'exécution spécifiques.**

---

<sup>a</sup> *footprint* ou empreinte mémoire.

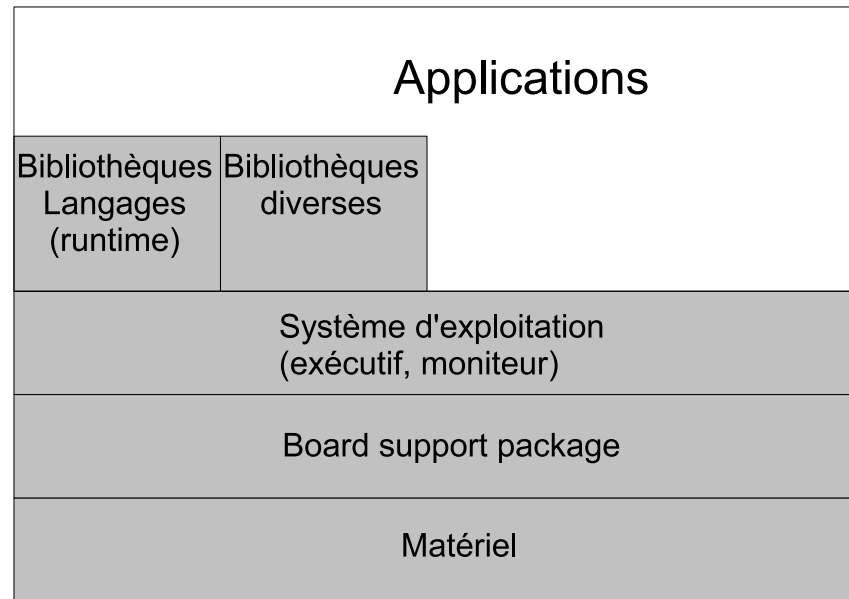
# Systemes d'exploitation temps réel (1)

---

- **Caractéristiques :**

- Aussi appelé "Moniteur" ou "Exécutif".
- Modulaire et de petite taille. Flexible vis-à-vis de l'application.
- Accès aisé aux ressources physiques.
- Abstractions adaptées (parallélisme, exception, interruption, tâches, ...)
- Support de langages pour le temps réel (ex : C, Ada).
- Livré avec ses performances temporelles (en théorie).
- Améliorer la portabilité : architecture + standardisation (du langage de programmation, des services du système d'exploitation).

# Systemes d'exploitation temps réel (2)



- **Architecture en couches :**

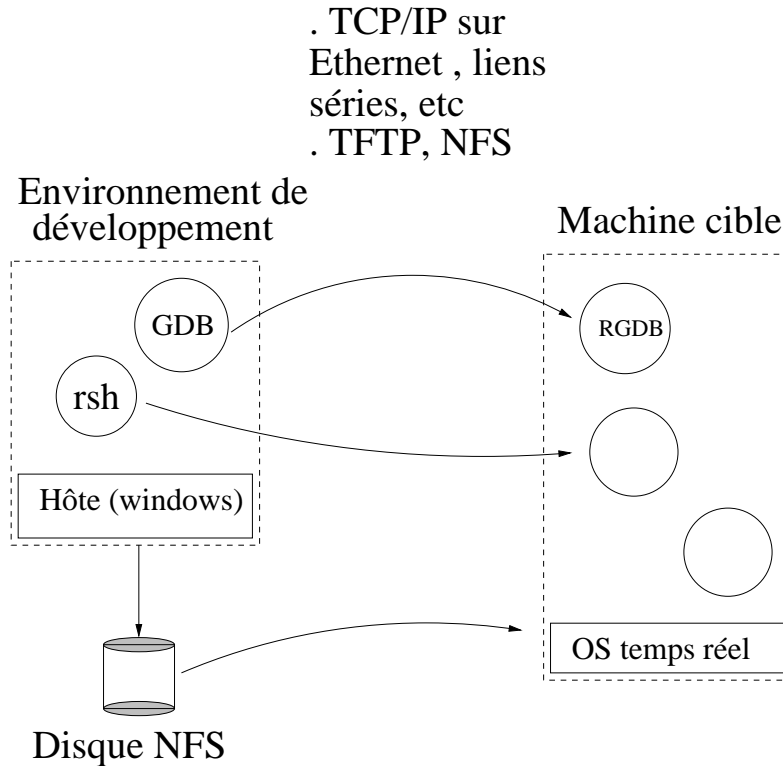
- Bibliothèque langage (ou runtime) constituant l'environnement d'exécution d'un programme (C, Ada). Portabilité de l'application (adapte le langage au système d'exploitation).
- BSP/Board support package : portabilité du système d'exploitation (adapte le système d'exploitation au matériel).

# Systemes d'exploitation temps réel (3)

---

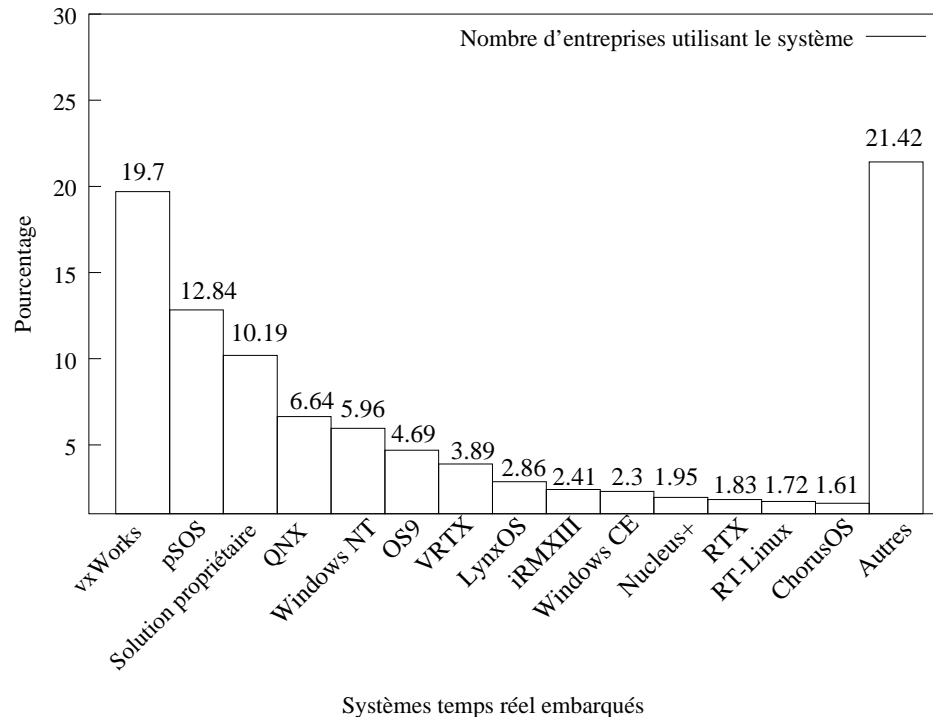
- **Performance connue et déterministe:**
  - Doit permettre l'évaluation de la capacité des tâches par exemple.
  - Utilisation de benchmarks (ex : *Rhealstone*, *Hartstone*, etc).
- **Critères de performances :**
  - Latence sur interruption.
  - Latence sur commutation de contexte/tâche.
  - Latence sur préemption.
  - Sémaphore "shuffle" (temps écoulé entre la libération d'un sémaphore et la réactivation d'une tâche bloquée sur celui-ci).
  - Temps de réponse pour chaque service (appel système, fonctions de bibliothèque).
  - etc.

# Systemes d'exploitation temps réel (4)



- **Phase de développement** : édition du source, compilation croisée, téléchargement, exécution et tests.
- **Phase d'exploitation** : construction d'une image minimale (exécutif + application) sans les services de développement. Stockage en EEPROM, Flash.

# Etat du marché (1)



- **Caractéristiques du marché [TIM 00] :**

- Diversité des produits présents  $\implies$  produits généralistes ou spécifiques à des applications types.
- Présence importante de produits "maisons".



# Etat du marché (2)

---

- **Quelques exemples de produits industriels :**

- VxWorks : produit généraliste et largement répandu (PABX, terminal X de HP, Pathfinder, satellite CNES, etc).
- pSOS édité par ISI (appli militaire, tél. portable).
- VRTX édité par Microtec (appli militaire, tél. portable).
- LynxOs (Unix temps réel).
- Windows CE/Microsoft (systèmes embarqués **peu** temps réel).

- **Produits "open-source" :**

- OSEK-VDX (appli. automobile).
- RTEMS de Oar (appli. militaire).
- eCos de cygnus.
- RT-Linux.

# Etat du marché (3)

---

- **Quelques standards :**
  - Langages de conception logicielle: UML/MARTE, AADL, HOOD HRT, ...
  - Langages de programmation : Ada 2005, C, ...
  - Systèmes d'exploitation : POSIX, ARINC 653, OSEK VDX, ...

# La norme POSIX (1)

---

- **Objectif** : définir une interface standard des services offerts par UNIX[VAH 96, J. 93] afin d'offrir une certaine **portabilité** des applications.
- Norme publiée conjointement par l'ISO et l'ANSI.
- **Problèmes** :
  - Portabilité difficile car il existe beaucoup de différences entre les UNIX.
  - Tout n'est pas (et ne peut pas ?) être normalisé.
  - Divergence dans l'implantation des services POSIX (ex : threads sur Linux).
  - Architecture de la norme.
- **Exemple de systèmes POSIX** : Lynx/OS, VxWorks, Solaris, Linux, QNX, etc .. (presque tous les systèmes temps réel).

# La norme POSIX (2)

---

- **Architecture de la norme** : découpée en chapitres optionnels et obligatoires. Chaque chapitre contient des parties obligatoirement présentes, et d'autres optionnelles.
- Exemple de chapitres de la norme POSIX :

Chapitres	Signification
POSIX 1003.1	Services de base (ex : <i>fork</i> , <i>exec</i> , ect)
POSIX 1003.2	Commandes shell (ex : <i>sh</i> )
POSIX 1003.1b [GAL 95]	Temps réel
POSIX 1003.1c [RIF 95]	Threads
POSIX 1003.5	POSIX et Ada
etc	

# La norme POSIX (3)

---

- Cas du chapitre POSIX 1003.1b : presque tout les composants sont optionnels !!

Nom	Signification
_POSIX_PRIORITY_SCHEDULING	Ordonnancement à priorité fixe
_POSIX_REALTIME_SIGNALS	Signaux temps réel
_POSIX_ASYNCHRONOUS_IO	E/S asynchrones
_POSIX_TIMERS	Chien de garde
_POSIX_SEMAPHORES	Sémaphores
etc ...	

- **Conséquence** : que veut dire "être conforme POSIX 1003.1b" ... pas grand chose puisque la partie obligatoire n'est pas suffisante pour construire des applications temps réel.

# La norme POSIX (4)

---

- Les threads POSIX.
- Services d'ordonnancement.
- Outils de synchronisation.
- Les signaux temps réel.
- La manipulation du temps.
- Les entrées/sorties asynchrones.
- Les files de messages.
- La gestion mémoire.

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.

# Introduction au langage Ada 2005 (1)

---

- **Pourquoi ce langage :**

- Abstractions temps réel : tâche, interruption, ordonnancement, synchronisation, timer et gestion du temps, ...
- Langage standardisé par l'ISO (portabilité).
- Compilation séparée (logiciel volumineux) et typage fort (fiabilité).
- Nombreuses analyses statiques. Pas de dynamisme. Pas d'allocation dynamique. Pas de dépendance cyclique.
- Langage complexe.

- **Domaines** : transport (avionique et ferroviaire), spatial, militaire.

- **Exemples** : Airbus (320, 380), Boeing (777), Fokker, Tupolev, Eurostar, Metro (14 Paris), TGV, Ariane (4 et 5), Satellites (Intersat), spatial (sonde Cassini, Huygens, Soho, Mars Express), militaire (Tigre, Apache, Patriot)

⇒ <http://www.seas.gwu.edu/mfeldman/ada-project-summary.html>.



# Introduction au langage Ada 2005 (2)

---

1. C'est quoi un programme Ada ?
2. Types, opérateurs, variables, constantes.
3. Structure et flot de contrôle.
4. Entrées/sorties.
5. Pointeur et allocation dynamique.
6. Paquetages génériques.

# C'est quoi un programme Ada (1)

---

- **Compilation séparée** : unité de programme = entité logicielle compilable indépendamment. Logiciels de grande taille.
- **Catégories d'unité de programme (fichiers GNAT) :**
  - **Procédure principale** : point d'entrée d'un programme (fichier .adb).
  - **Paquetage** : collection de déclarations (sous programmes, types, tâches, ...).
    - Partie visible (spécification du paquetage, fichier .ads).
    - Partie cachée (corps du paquetage, fichier .adb).
  - **Tâches** : spécification (fichier .ads) et implémentation (fichier .adb).
  - **Unités génériques**: unités (paquetage ou sous programmes) paramétrées par types, constantes, sous-programmes ou paquetages.

# C'est quoi un programme Ada (2)

---

- **Structure d'une procédure principale :**

```
with nom_paquetage1; use nom_paquetage1;  
with nom_paquetage2; use nom_paquetage2;
```

```
procedure nom_procedure_principale is  
— declarations  
begin  
— instructions  
end nom_procedure_principale;
```

- Fichier `nom_procedure_principale.adb`
- Clauses `with et use`
- `Use` est optionnel => fiabilité. Notation pointée sinon.

# C'est quoi un programme Ada (3)

---

- **Exemple de procédure principale :**

```
with text_io ;  
use text_io ;  
  
procedure Coucou is  
begin  
    Put_Line ( " Coucou " );  
end Coucou ;
```

# C'est quoi un programme Ada (4)

---

- **Structure d'une spécification de paquetage :**

```
package nom_du_paquetage is
— declarations publiques
private
— declarations privees
end nom_du_paquetage;
```

- **Structure d'une implémentation de paquetage :**

```
package body nom_du_paquetage is
— sous programmes
begin
— code d'initialisation
end nom_du_paquetage;
```

# C'est quoi un programme Ada (5)

---

- **Spécification d'un paquetage (fichier `lemien.ads`) :**

```
package Lemien is
```

```
    procedure somme(a : in integer;
```

```
        b : in integer; resultat : out integer);
```

```
    function somme(a : in integer; b : in integer)
```

```
        return integer;
```

```
private
```

```
    variable_interne : integer;
```

```
end Lemien;
```

- Surchage => fiabilité.
- Contrôle arguments `in/out` => fiabilité.

# C'est quoi un programme Ada (6)

---

- **Implémentation d'un paquetage (lemien.adb) :**

```
package body Lemien is
```

```
    procedure somme(a : in integer;  
                  b : in integer; resultat : out integer) is
```

```
begin
```

```
    resultat:=a+b+variable_interne;
```

```
end somme;
```

```
function somme(a : in integer; b : in integer)
```

```
    return integer is
```

```
begin
```

```
    return a+b+variable_interne;
```

```
end somme;
```

```
begin
```

```
    variable_interne:=100;
```

```
end Lemien;
```

# C'est quoi un programme Ada (7)

---

- **Exemple d'utilisation du paquetage Lemien :**

```
with text_io;  
use text_io;  
with Lemien;  
use Lemien;  
  
procedure princ is  
a : integer :=0;  
begin  
    somme(10,20,a);  
    Put_Line(integer'image(a));  
    a:=somme(40,50);  
    Put_Line(integer'image(a));  
end princ;
```



# C'est quoi un programme Ada (8)

---

- **Compiler ce programme (avec GNAT) :**

```
>gnatmake princ.adb
```

```
gcc -c lemien.adb
```

```
gcc -c princ.adb
```

```
gnatbind -x princ.ali
```

```
gnatlink princ.ali
```

- **gnatmake** : gestion des dépendances entre unités de programme.
- **gcc** : compilation
- **gnatbind** : phase d'élaboration (initialisation des paquetages).
- **gnatlink** : édition des liens.
- **Résultats** : `princ`, `lemien.ali`, `lemien.o`, `princ.ali` et `princ.o`

# C'est quoi un programme Ada (9)

---

- **Exercice 1 :**

```
package calcul is
    function additionne(a : in integer; b : in integer)
        return integer;
    function multiplie(a : in integer; b : in integer)
        return integer;
    function soustrait(a : in integer; b : in integer)
        return integer;
    function divise(a : in integer; b : in integer)
        return integer;
end calcul;
```

Écrire une procédure principale qui , grâce au paquetage `calcul`, calcule et affiche l'expression suivante  $(2 \cdot 3) + 4$ . Écrire l'implémentation du paquetage.

# Types, opérateurs, variables (1)

---

- **Typage fort:**

- Améliorer la maintenabilité (lisibilité).
- Améliorer la sécurité: analyse statique à la compilation et à l'exécution (exception).
- Interdire les opérations entre variables de types différents (pas de cast implicite).

- **Type:**

- Type = taille mémoire + représentation + plage de valeurs + attributs/opérateurs.
- Plage de valeurs définie par la norme (portabilité).
- Attributs : opérateurs pré-définis pour tous les types, définis par l'utilisateur ou non.

# Types, opérateurs, variables (2)

---

- **Types scalaires :**

- `float`, `integer`, `boolean`, `character`, `access` ainsi que les énumérations.
- Exemples d'attribut : `integer'last`, `integer'first`, `integer'range`

- **Types composés :** `array`, `string` (qui est un `array`), `record`, `union`, `task`, `protected`

- **Principaux opérateurs :**

- Arithmétiques : `+`, `-`, `*`, `/`, `mod`
- Relationnels : `=`, `/=`, `<=`, `>=`, `in`, `not`, `and`, `or`, `xor`

# Types, opérateurs, variables (3)

---

- **Types dérivés** : si  $a$  est un type dérivé de  $b$ , alors  $a$  et  $b$  sont deux types différents, et ne sont pas compatibles.
- **Sous types** : si  $a$  est un sous type de  $b$ , alors  $a$  et  $b$  sont compatibles.  $a$  est un alias de  $b$ .

# Types, opérateurs, variables (4)

---

- **Exemples de déclarations :**

```
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, opérateurs, variables (5)

---

- **Exemples types dérivés et sous 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, opérateurs, variables (6)

---

- **Exemples types dérivés et sous 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, opérateurs, variables (7)

---

- Le typage fort facilite l'analyse statique.
- Exemple du programme C de D. Lesens [LES 10].

```
// Programme C incorrect
// qui compile correctement
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, opérateurs, variables (8)

---

- Et la version Ada maintenant :

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

— Programme Ada incorrect

— qui NE compile PAS

```
procedure cfaux 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 cfaux;
```

# Types, opérateurs, variables (9)

---

- **Types composés :**

1. **Constructeurs de type :** `type`
2. **Énumération :** type discret, implantation mémoire cachée (ex: `enum` en C) mais attributs spécifiques (`succ` et `pos`).
3. **Structure :** constructeur `record`. Initialisation des attributs par ordre de déclaration ou en les nommant.
4. **Tableau :** constructeur `array`, une ou deux dimensions, indices de type discret (entier, énumération), taille connue à la définition du type ou à la déclaration du tableau.

# Types, opérateurs, variables (10)

---

- **Exemple d'énumération :**

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

type un_jour is (lundi , mardi , mercredi , jeudi ,
    vendredi , samedi , dimanche);
j : un_jour := lundi;

package io is new text_io.enumeration_io(un_jour);

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

# Types, opérateurs, variables (11)

---

- **Exemple de tableau :**

```
type un_jour is (lundi , mardi , mercredi , jeudi ,  
vendredi , samedi , dimanche);
```

```
type tab1 is array (0..3) of integer;
```

```
type tab2 is array (1..4) of un_jour;
```

```
type tab3 is array (lundi .. dimanche) of integer;
```

```
t1 : tab1 := (30,43,28,100);
```

```
t2 : tab2 := (4=>lundi , 2=>mardi ,  
3=>dimanche , 1=>mercredi );
```

```
t3 : tab3;
```

```
begin
```

```
  t1 (0):= t1 (0)*2;
```

```
  t2 (1):= dimanche;
```

```
  t3 (lundi ):=2;
```

```
  ...
```

# Types, opérateurs, variables (12)

---

- **Exemple de structure :**

```
with text_io;
use text_io;
procedure point is

type un_point is record
    x : integer;
    y : integer;
end record;

p1 : un_point := (10,20);
p2 : un_point := (y=>20, x=>10);

begin
    Put_Line(integer'image(p1.x));
    Put_Line(integer'image(p1.y));
end point;
```

# Types, opérateurs, variables (13)

---

- **Exercice 2** : pour chaque affectation, indiquez si elle est correcte ou non.

```
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;
```

# Flots de contrôle (1)

---

- Séquence :

```
i1 ; i2
```

- Conditionnelle :

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



# Flots de contrôle (2)

---

- Diverses formes d'itération :

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

```
loop
  i1 ; i2 ;
  exit cond ;
end loop ;
```

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

# Flots de contrôle (3)

---

- **Exemple employant des attributs:**

```
s1,s2,s3 : integer:=0;
```

```
subtype indice is integer range 1..10;
```

```
...
```

```
for i in 1..10 loop
```

```
    s1:=s1+i;
```

```
end loop;
```

```
for j in indice'first..indice'last loop
```

```
    s2:=s2+j;
```

```
end loop;
```

```
for k in indice'range loop
```

```
    s3:=s3+k;
```

```
end loop;
```

# Entrées/sorties (1)

---

- **Typage fort** : chaque type doit disposer des services d'entrées/sorties mais familles de type.
- **Services offerts par le paquetage `Text_Io`** : pour les types `String` et `Character` uniquement (extrait de GNAT):
  - `Get` : saisie d'une chaîne de caractères de taille fixe.
  - `Put` : affichage d'une chaîne de caractères.
  - `New_Line` : retour chariot
  - `Put_Line` : `Put` + `New_Line`
  - `Get_Line` : saisie d'une chaîne de caractères de taille variable.
- **Autres types** : instancier les paquetages génériques `Float_Io`, `Integer_Io`, `Enumeration_Io`, ...

# Entrées/sorties (2)

---

- **Spécification de 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 ...
```

# Entrées/sorties (3)

---

- Exemple du générique `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;
```

# Entrées/sorties (4)

---

- Exemple d'utilisation de `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(" Saisir temperature 1:");
    temperature_io.Get(t1);
    New_Line; Put(" Saisir temperature 2:");
    temperature_io.Get(t2);
    New_Line;
    Put("Somme = "); temperature_io.Put(t1+t2);
    New_Line;
exception
    when Data_Error =>
        Put_line("Donnee saisie non conforme au type 'temperature'");
end Intio;
```

# Entrées/sorties (5)

---

- **Exercice 3 :**

Écrire un programme qui permet de saisir des entiers et affiche la somme des valeurs saisies au fur et à mesure des saisies. Le programme doit afficher une erreur lorsque les données saisies ne sont pas entières.

# Pointeurs, allocations dynamiques (1)

---

- Généralement pas de pointeur et pas d'allocation dynamique dans les systèmes temps réel, mais :

- **Typage fort** : un pointeur ne peut adresser qu'un seul type de donnée. Pointeur typé.

- **Contrôle sur l'utilisation des pointeurs** : fiabilité.

- **Exemple de déclarations** :

```
type Integer_Ptr is access Integer;  
pointeur1 : Integer_Ptr := null;  
mon_integer : Integer;  
pointeur2 : Integer_Ptr := mon_integer'access;
```

- **Allocation dynamique** : opérateur `new`.

- **Désallocation** : non standardisée.



# Pointeurs, allocations dynamiques (2)

---

- Exemple :

```
with Text_lo; use Text_lo;
procedure Pointeur is
    package lo is new Text_lo.Integer_lo(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);
    lo.Put(P1.all);
    lo.Put(P2.all);
    lo.Put(P4.all);
    lo.Put(P3.all);
end Pointeur;
```

# Pointeurs, allocations dynamiques (3)

---

- **Contrôle sur l'utilisation des pointeurs** : fiabilité.

```
with Text_lo; use Text_lo;
procedure Alloc_Faux is

    type Integer_Ptr is access Integer;
    Global : Integer_Ptr;

    procedure Affecter_Valeur is
        I1 : Integer := 100;
    begin
        Global := I1 'access;
    end Affecter_Valeur;

    package lo is new Text_lo.Integer_lo(Integer);
begin
    Affecter_Valeur;
    lo.Put(Global.all);
end Alloc_Faux;
```

# Unités de programme génériques (1)

---

- **Unité de programme paramétrée par** : types, constantes, sous-programmes, paquetages.
- Procédure ou paquetage générique.
- Permet d'effectuer un traitement identique sur plusieurs entités différentes (ex : types).
- **Instanciation** : une unité de programme générique ne peut pas être employée sans être **instanciée**, c-à-d donner une valeur pour chaque paramètre.
- **Structure** :
  - generic
  - paramètres du générique
  - package foo ...
  - package body foo ...
  - utilisation des paramètres dans la spécification
  - et l'implémentation du générique

# Unités de programme génériques (2)

---

```
generic
  type Element is private;
  with procedure Put(E : in Element);
package Listes is
  type Element_Ptr is access Element;
  type Cellule is private;
  type Lien is access Cellule;

  procedure Afficher(L : in Lien);
  procedure Ajouter(L : in out Lien; E : in Element_Ptr);
private
  type Cellule is record
    Suivant : Lien;
    Info    : Element_Ptr;
  end record;
end Listes;
```

[MCours.com](http://MCours.com)

# Unités de programme génériques (3)

---

```
package body Listes is
  procedure Ajouter (L : in out Lien; E : in Element_Ptr) is
    Nouveau : Lien;
  begin
    Nouveau:=new Cellule;
    Nouveau.Info:=E; Nouveau.Suivant:=L; L:=Nouveau;
  end Ajouter;

  procedure Afficher (L : in Lien) is
    Courant : Lien := L;
  begin
    while Courant/=null loop
      Put(Courant.Info.all);
      Courant:=Courant.Suivant;
    end loop;
  end Afficher;
end Listes;
```

# Unités de programme génériques (4)

---

```
with Listes ;
procedure Teste_Listes is

    type Personne is record ...
    procedure Affiche (A : in Personne) is ...

package Ma_Liste is new Listes(Personne, Affiche);
use Ma_Liste;

Une_Liste : Lien;
P          : Ma_Liste.Element_Ptr;  — pointeur sur
                                     — une personne

begin
    P:= new Personne;
    Ajouter(Une_Liste, P);
    Afficher(Une_Liste);
    ...
```

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.

# Concurrence

---

- Tâche
- Synchronisation et communication par rendez vous.
- Communication par objets protégés.



# Tâche (1)

---

- **Une tâche Ada est constituée :**
  - D'une spécification qui décrit son interface. Partie visible.
  - D'une implémentation qui contient le code exécuté par la tâche. Partie cachée.
  - Éventuellement d'un type. Dans le cas contraire, on parle de tâche anonyme.
- **Une tâche Ada est déclarée par :**
  - Les instructions `task/task type` (spécification de la tâche) et `task body` (implantation de la tâche).
  - La procédure principale est aussi une tâche.

# Tâche (2)

---

- **Exemple d'une tâche anonyme, allouée statiquement :**

```
with Text_io; use Text_io;
procedure Tache_anonyme is
  task Ma_Tache;
  task body Ma_Tache is
  begin
    loop
      Put_Line("tache activee");
      delay 1.0;
    end loop;
  end Ma_Tache;
begin
  null;
end Tache_anonyme;
```

- Combien de tâches ici ?

# Tâche (3)

---

- **Exemple de tâches typées, allouées statiquement :**

```
with Text_io; use Text_io;
procedure Tache_type is
  task type Un_Type;
  task body Un_Type is
  begin
    loop
      Put_Line("tache activee");
      delay 1.0;
    end loop;
  end Un_Type;
  T1, T2: Un_Type;
  T : array (1..10) of Un_Type;
begin
  null;
end Tache_type;
```

- **Combien de tâches ici ?**

# Tâche (4)

---

- **Exemple de tâches typées, allouées dynamiquement :**

```
with Text_io; use Text_io;
procedure Tache_dynam is
  task type Un_Type;
  task body Un_Type is
  begin
    loop
      Put_Line("tache activee");
      delay 1.0;
    end loop;
  end Un_Type;
  type Un_Type_Ptr is access Un_Type;
  T : array (1 .. 3) of Un_Type_Ptr;
begin
  for i in 1..3 loop
    T(i):= new Un_Type;
  end loop;
end Tache_dynam;
```

# Tâche (5)

---

- **Une tâche peut être** : active, avortée, achevée, terminée.
- **Règles d'activation** :
  - Tâche allouée statiquement: au début de bloc où la tâche est définie.
  - Tâche allouée dynamiquement : lors de l'allocation dynamique.
- **Règles de terminaison** :
  - Sur exception: l'exception est perdue si non rattrapée.
  - Lorsque toutes les tâches de niveau inférieur sont achevée.
- **Avortement** : grâce à l'instruction `abort x`, avec `x` le nom de la tâche.

# Tâche (6)

---

- Ce programme est faux. Pourquoi ?

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

# Tâche (7)

---

- **Exercice 4 :**

Dites pour les exemples de programme des pages 58, 59 et 60 quand les tâches sont activées et quand elles sont terminées.

# Tâche (8)

---

- **Exercice 5 :**

Écrire un programme qui contient deux tâches. La première tâche calcule et affiche les éléments de la suite  $U_n = U_{n-1} * 2$  avec  $U_0 = 1$ . La tâche doit attendre une seconde entre le calcul/affichage de deux éléments successifs. La deuxième tâche calcule et affiche les éléments de la suite  $U_n = U_{n-1} + 2$  avec  $U_0 = 0$ . La tâche doit attendre deux secondes entre le calcul/affichage de deux éléments successifs.

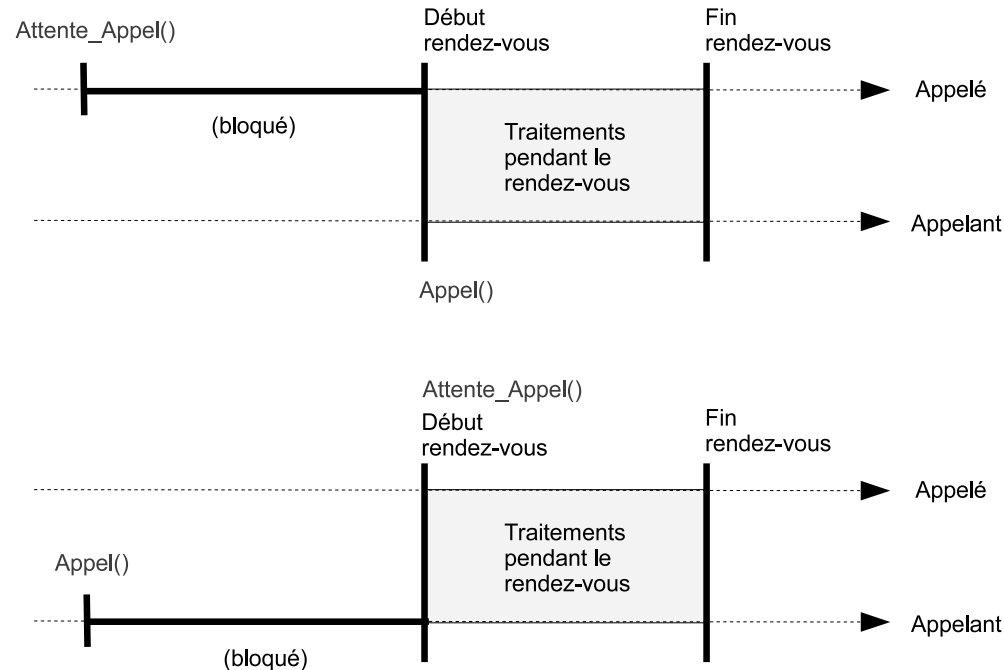


# Concurrence

---

- Tâche
- Synchronisation et communication par rendez vous.
- Communication par objets protégés.

# Mécanisme de rendez vous (1)



- **Le rendez vous est un mécanisme :**

- Asymétrique : tâche appelante et tâche appelée.
- Qui permet de synchroniser deux tâches : le rendez vous nécessite que les deux tâches soient simultanément prêtes.
- Qui permet l'échange de données entre deux tâches.

# Mécanisme de rendez vous (2)

---

- **Mise en œuvre des rendez vous avec Ada :**
  - **Notion d'entrée:** point de rendez vous déclaré dans la spécification de la tâche. Interface de la tâche.
  - **Spécification de tâche :** peut comporter plusieurs entrées.
  - **Instruction accept :** permet à une tâche d'attendre un rendez vous sur l'une de ses entrées, puis, d'exécuter une séquence d'instructions pendant le rendez vous.
  - **Appelants :** se placent en position de rendez vous en référénçant l'entrée voulue.

# Mécanisme de rendez vous (3)

---

- **Exemple de rendez vous (synchronisation uniquement):**

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

# Mécanisme de rendez vous (4)

---

- **Rendez vous avec communication:**

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

# Mécanisme de rendez vous (5)

---

- Quelques clauses additionnelles :
  - **Clause `select`** : permet d'attendre simultanément sur plusieurs entrées.
  - **Clause `terminate`** : utilisée conjointement avec `select`, permet d'interrompre proprement une instruction `accept`
  - **Entrée gardée** : condition booléenne d'éligibilité de l'entrée.
  - **Clause `else`** : attente non bloquante sur les entrées.
  - ...

# Mécanisme de rendez vous (6)

---

- **Tâches avec plusieurs entrées :**

```
task body Ma_Tache 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 Ma_Tache;
```

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

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

# Mécanisme de rendez vous (7)

---

- **Exemple avec la clause terminate :**

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

```
task Ma_Tache is
  entry Incrementer
    (S1 : in out Integer);
end Ma_Tache;

Val : Integer :=0;

begin
  Ma_Tache.Incrementer(Val);
  Put_line("Val = " &
    integer'image(Val));
end increment_terminate;
```



# Concurrence

---

- Tâche
- Synchronisation et communication par rendez vous.
- Communication par objets protégés.

# Objets et types protégés (1)

---

- **Un objet protégé c'est :**
  - Une structure de données comportant plusieurs variables accessibles de façon concurrente.
  - Un mécanisme de synchronisation proche du modèle lecteur-rédacteur.
- **Un objet protégé est constitué :**
  - D'une spécification qui décrit son interface: procédures, fonctions et entrées. Partie visible.
  - D'une implémentation qui inclue les variables protégées et le code des fonctions, procédures et entrées. Partie cachée.
  - Éventuellement d'un type. Dans le cas contraire, on parle d'objet anonyme.

# Objets et types protégés (2)

---

- **Verrous mis en œuvre par un objet protégé :**
  - **Fonctions** : peuvent être exécutées de façon concurrente car ne change pas l'état des variables protégées : lecture des variables seulement.
  - **Procédures** : exécutées en exclusion mutuelle sur les variables protégées.  
  
⇒ Verrous fonctions et procédures = lecteurs/rédacteurs.
  - **Entrées** : idem procédure + garde booléenne. Une entrée dont la garde est à `false` ne peut pas être exécutée (blocage de la tâche), même si aucune procédure/entrée/fonction utilise l'objet.

# Objets et types protégés (3)

---

- **Exemple d'une variable protégée (spécification) :**

```
package Vars is
  protected type Var is
    procedure Ecriture(Valeur : in Integer);
    function Lecture return Integer;
  private
    Variable : Integer:=0;
  end Var;
end Vars;
```

# Objets et types protégés (4)

---

- **Exemple d'une variable protégée (corps) :**

```
package body Vars is
  protected body Var is
    procedure Ecriture(Valeur : in Integer) is
    begin
      Variable := Valeur;
    end Ecriture;
    function Lecture return Integer is
    begin
      return Variable;
    end Lecture;
  end Var;
end Vars;
```

# Objets et types protégés (5)

---

- **Exemple d'une variable protégée (utilisation) :**

```
with Vars; use Vars;
procedure Variable_Protegee is
  Une : Vars.Var;
  task Ma_Tache;
  task body Ma_Tache is
  begin
    loop
      Put_Line("Val = " & integer'image(Une.Lecture));
    end loop;
  end Ma_Tache;
  I : Integer :=0;
begin
  loop
    Une.Ecriture(I);
    I:=I+1;
  end loop;
end Variable_Protegee;
```

# Objets et types protégés (6)

---

- **Exemple d'un sémaphore (spécification):**

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

# Objets et types protégés (7)

---

- **Exemple d'un sémaphore (corps):**

```
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;
```



# Objets et types protégés (8)

---

```
Mutex : Semaphore;  
task type Une;  
task body Une is  
begin  
    loop  
        Mutex.P;  
        Put_Line("en section critique !!");  
        Mutex.V;  
    end loop;  
end Une;  
type Une_Ptr is access Une;  
Plusieurs : array (1..10) of Une_Ptr;  
  
begin  
    Mutex.Init(1);  
    for i in 1..10 loop  
        Plusieurs(i):= new Une;
```

# Objets et types protégés (9)

---

- **Exemple d'un lecteur/rédacteur :**

```
package Lecteurs_Redacteurs is
  protected type Lecteur_Redacteur is
    entry Debut_Lecture;
    procedure Fin_Lecture;
    entry Debut_Ecriture;
    procedure Fin_Ecriture;
  private
    Nb_Lecteurs : Natural :=0;
    Nb_Redacteurs : Natural :=0;
  end Lecteur_Redacteur;
end Lecteurs_Redacteurs;
```

# Objets et types protégés (10)

---

```
protected body Lecteur_Redacteur is
  entry Debut_Lecture when Nb_Redacteurs = 0 is
  begin
    Nb_Lecteurs:=Nb_Lecteurs+1;
  end Debut_Lecture;
  entry Debut_Ecriture when Nb_Lecteurs + Nb_Redacteurs = 0 is
  begin
    Nb_Redacteurs:=Nb_Redacteurs+1;
  end Debut_Ecriture;
  procedure Fin_Lecture is
  begin
    Nb_Lecteurs:=Nb_Lecteurs-1;
  end Fin_Lecture;
  procedure Fin_Ecriture is
  begin
    Nb_Redacteurs:=Nb_Redacteurs-1;
  end Fin_Ecriture;
end Lecteur_Redacteur;
```

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.

# Temps réel

---

- **Services temps réel disponibles via les standards:**
  - ISO/IEC Ada 1995/2005 : et en particulier les annexes C (Systems programming) et D (Real-Time) [TAF 06].
  - "Binding" Ada POSIX 1003 [BUR 07, GAL 95].
  - ARINC 653 [ARI 97].
  - ...

# Temps réel et Ada 2005

---

- **Avec Ada 2005, services temps réel offerts via des pragmas et des paquetages spécifiques:**
  - Comment implanter une tâche périodique :
    1. Représenter le temps (paquetage `Ada.Real_Time`).
    2. Implanter les réveils périodiques (instruction `delay`).
    3. Affecter des priorités (pragma).
  - Comment activer un protocole d'héritage de priorité (pour des objets/types protégés).
  - Comment sélectionner un ordonnanceur (RM, EDF, ...).
  - ...

# Ada 2005: tâches périodiques (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 (NS : Integer) return Time_Span;
  function Seconds (S : Integer) return Time_Span;
  function Minutes (M : Integer) return Time_Span;
  ...
```

- `Ada.Real_Time` est un paquetage qui offre une horloge **haute résolution** and **documentée**, ainsi que des sous-programmes pour la manipuler.

# Ada 2005: tâches périodiques (2)

---

- **Principaux types et sous-programmes :**
  - `Time` implémente un date absolue. Ce type doit permettre d'exprimer des dates de 50 ans au moins.
  - `Time_Span` représente une durée.
  - `Time_Unit` la plus petite unité de temps représentable par le type `Time`. Défini par l'implémentation, mais doit être inférieur ou égal à 20 micro-secondes.
  - `Clock` retourne le temps écoulé depuis *epoch*.
  - Sous-programmes pour convertir des données temporelles en `Time_Span` : `Nanoseconds`, `Microseconds`, ...



# Ada 2005: tâches périodiques (3)

---

- L'instruction `delay` :

1. `delay expr` : bloque une tâche pendant **au moins** `expr` unités de temps.

2. `delay until expr` : bloque une tâche jusqu'**au moins** la date `expr`.

- Une tâche ne peut pas être réveillée **avant le délai/date** spécifié par l'instruction `delay`.

- Mais une tâche peut très bien être réveillée **après le délai/date** spécifié par l'instruction `delay`. Une borne maximale doit être documentée par l'éditeur du compilateur.

# Ada 2005: tâches périodiques (4)

---

- **Exemple d'une tâche périodique :**

```
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
        Read_Speed;
        Next_Time := Next_Time + Period;
        delay until Next_Time;
    end loop;
end Tspeed;
```

# Ada 2005: tâches périodiques (5)

---

```
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;
  ...
```

- `System.Priority` doit offrir au moins 30 niveaux de priorité, mais plus, c'est mieux pour l'analyse de l'ordonnancement.
- Priorité de **Base** = priorité affectée statiquement.
- Priorité **active** = priorité héritée (accès aux objets protégés).

# Ada 2005: tâches périodiques (6)

---

- **Règles d'assignation des priorités avec Ada 2005:**
  - Toute tâche a une priorité par défaut (voir le paquetage `System`).
  - Le pragma `Priority` peut être employé dans la spécification d'une tâche.
  - Le pragma `Priority` peut être employé dans une procédure principale.
  - Sans pragma, toute tâche hérite de la priorité de la tâche qui l'a instanciée.

# Ada 2005: tâches périodiques (7)

---

- **Déclaration d'une tâche :**

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

- **Déclaration d'un type tâche :**

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

- **Déclaration d'un type tâche avec discriminant :**

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

# Ada 2005: tâches périodiques (8)

- Soit le jeu de tâches :

Tache	Période (milli-secondes)	Priorité
$T_{display}$	$P_{display} = 100$	12
$T_{engine}$	$P_{engine} = 500$	10
$T_{speed}$	$P_{speed} = 250$	11

- Et le code de chaque tâche :

```
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: tâches périodiques (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: tâches périodiques (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: tâches périodiques (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 ("Les tâches démarrent sur la terminaison de
           la procédure principale");
end Car_System;
```

# Ada 2005: types/objets protégés (1)

---

- **Protocole d'héritage de priorités supportés par Ada 2005** : ICPP (Immediate Ceiling Priority Protocol) et PLCP (Preemption Level Control Protocol).
- **ICPP est une variation de PCP dont le fonctionnement est:**
  - Priorité plafond d'un objet protégé = maximum des priorités de base des tâches qui l'emploient.
  - Priorité active d'une tâche = maximum entre sa priorité de base et la priorité plafond de tous les objets protégés qu'elle a verrouillés.

# Ada 2005: types/objets protégés (2)

---

- **Affectation d'une priorité plafond à un objet protégé :**

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

- **Pour activer le protocole ICPP :**

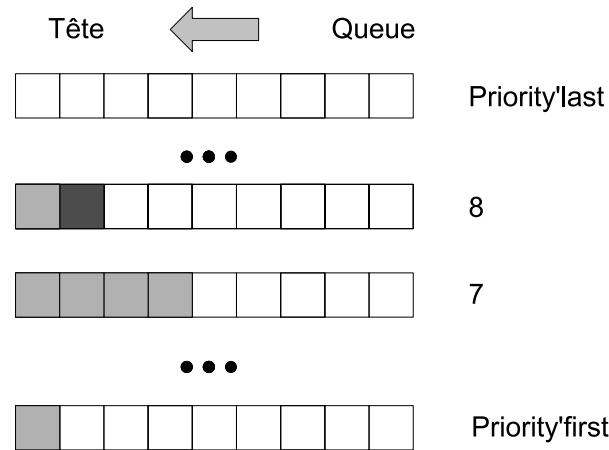
```
pragma Locking_Policy ( Ceiling_Locking );
```

# Temps réel et Ada 2005

---

- **Avec Ada 2005, services temps réel offerts via des pragmas et des paquetages spécifiques:**
  - Comment implanter une tâche périodique :
    1. Représenter le temps (paquetage `Ada.Real_Time`).
    2. Implanter les réveils périodiques (instruction `delay`).
    3. Affecter des priorités (pragma).
  - Comment activer un protocole d'héritage de priorité (pour des objets/types protégés).
  - Comment sélectionner un ordonnanceur (RM, EDF, ...).
  - ...

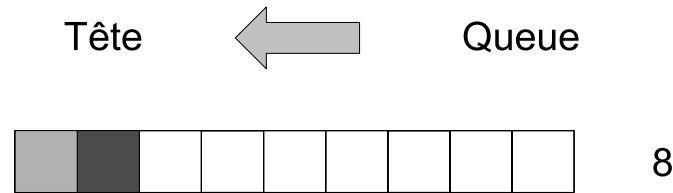
# Ordonnancement avec Ada 2005 (1)



- Une file d'attente pour chaque niveau de priorité. Toutes les tâches prêtes de même priorité sont insérées dans la même file d'attente.
- Chaque file d'attente a une politique (ou *dispatching policy*).
- Deux niveaux d'ordonnancement:
  1. Choisir la file d'attente de priorité maximale avec au moins une tâche.
  2. Choisir la tâche à exécuter parmi les tâches de la file d'attente sélectionnée en (1), et selon la politique de la file.

# Ordonnancement avec Ada 2005 (2)

---



- **Exemple de la politique d'ordonnancement préemptive à priorité fixe (politique `FIFO_Within_Priorities`) :**

- Quand une tâche devient prête, elle est insérée à la queue de file associée à sa priorité.
- La tâche en tête de file obtient le processeur quand sa file est la file de plus haute priorité avec une tâche prête.
- Quand la tâche en cours d'exécution passe à l'état bloquée ou terminée, alors elle quitte la file.

⇒ **Il est facile d'appliquer les tests de faisabilité** (cf. cours sur Rate Monotonic), si les priorités sont correctement affectées (priorités uniques et selon la période).

# Ordonnancement avec Ada 2005 (3)

---

- **Activation de la politique `FIFO_Within_Priorities` par :**

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

- **Autres politiques d'ordonnancement d'Ada 2005 :**

1. Ordonnancement non préemptif à priorités fixes :

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

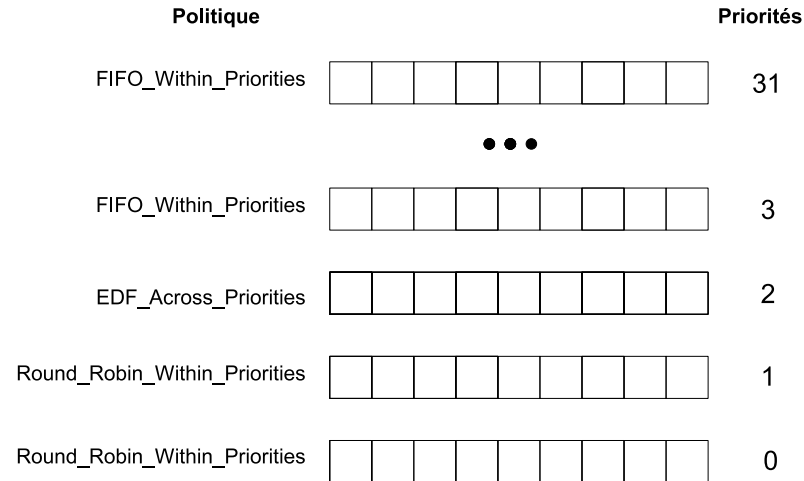
2. Ordonnancement Earliest deadline first :

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

3. Ordonnancement Round robin :

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

# Ordonnancement avec Ada 2005 (4)



- **Cohabitation de tâches critiques et non critiques grâce à des politiques différentes selon les niveaux de priorité :**

```
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);
```



# Ordonnancement avec Ada 2005 (5)

---

- Exemple de notre application automobile :

```
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;
```

— Fichier gnat.adc (ou directement dans l'unité de programme)

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

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

# Le profil Ravenscar (1)

---

- **Comment être certain que l'application peut être analysée avec les tests de faisabilité de Rate Monotonic ?  $\implies$  utiliser Ravenscar.**
- **Ravenscar** = sous-ensemble d'Ada compatible avec les tests de faisabilité.
- **Ravenscar** = profil défini par le standard Ada 2005.
- **Profil** = ensemble de restrictions, exprimées par des pragmas et vérifiées par le compilateur, et donc assurées à l'exécution.
- **Activation de Ravenscar :**  
`pragma profile (Ravenscar);`

# Le profil Ravenscar (2)

---

- **Restrictions de Ravenscar :**

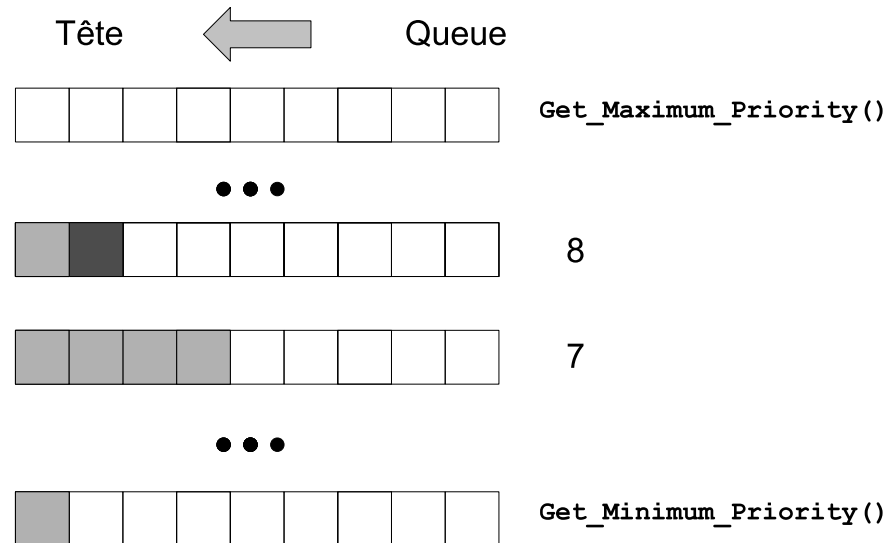
```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Restrictions (
    No_Abort_Statements ,
    No_Dynamic_Priorities ,
    No_Local_Protected_Objects ,
    No_Protected_Type_Allocators ,
    No_Requeue_Statements ,
    No_Specific_Termination_Handlers ,
    No_Task_Hierarchy ,
    Simple_Barriers ,
    Max_Entry_Queue_Length => 1 , Max_Protected_Entries => 1 ,
    Max_Task_Entries => 0 ,
    No_Dependence => Ada.Asynchronous_Task_Control ,
    No_Dependence => Ada.Calendar ,
    ...
);
```

# Temps réel

---

- **Services temps réel disponibles via les standards :**
  - ISO/IEC Ada 1995 and 2005 : et en particulier les annexes C (Systems programming) et D (Real-Time) [TAF 06].
  - **Binding Ada POSIX 1003 [BUR 07, GAL 95].**
  - ARINC 653 [ARI 97].
  - ...

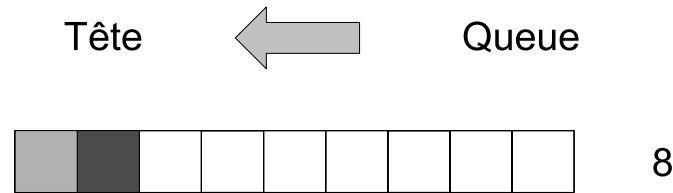
# Standard POSIX 1003 (1)



- Ordonnancement préemptif à priorités fixes. Au moins 32 niveaux de priorités. Files d'attente de tâches prêtes pour chaque priorité.
- Ordonnancement à deux niveaux :
  1. Choisir la file d'attente de priorité maximale avec au moins une tâche (prête).
  2. Choisir la tâche de la file sélectionnée en (1) selon une politique.

# Standard POSIX 1003 (2)

---



- **Politiques POSIX :**

1. `SCHED_FIFO` : identique à `FIFO_Within_Priorities`. La tâche quitte la tête de file si :
  - Terminaison de la tâche.
  - Blocage de la tâche (E/S, attente d'un délai) => remise en queue.
  - Libération explicite => remise en queue.
2. `SCHED_RR` : idem `SCHED_FIFO` mais en plus, la tâche en tête de file est déplacée en queue après expiration d'un quantum (`round robin`).
3. `SCHED_OTHERS` : fonctionnement non normalisé.

# Standard POSIX 1003 (3)

---

- Comment utiliser POSIX 1003.1b dans une application Ada : en utilisant le "binding" Ada POSIX 1003.5 (ex : Florist d'AdaCore).
- Florist est un ensemble de paquetage permettant de manipuler les concepts de priorité et politique POSIX :

```
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;
```

# Standard POSIX 1003 (4)

---

- **Sous-programmes qui permettent à l'application de s'adapter au système sous jacent :**

```
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;
```

```
...
```



# Standard POSIX 1003 (5)

---

- **Consultation/modification de la priorité d'un processus :**

```
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 ...
```

# Standard POSIX 1003 (6)

---

- **Exemple :**

```
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) ;
```

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

```
    ...
```

# Standard POSIX 1003 (7)

---

- **Doit on utiliser POSIX avec Ada ?**
- **Points positifs de POSIX:**
  - POSIX est supporté par de nombreux systèmes d'exploitation temps réel.
  - Les tests de faisabilité de Rate Monotonic sont utilisables avec POSIX (à condition que toutes les tâches aient des priorités uniques).
- **Points négatifs de POSIX :**
  - Un processus Unix, c'est quoi dans une runtime Ada ? Et les threads POSIX ?
  - POSIX est il portable ?

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.

# Exemples de runtimes Ada (1)

---

- **Rappels sur les runtimes :**

- Bibliothèques constituant l'environnement d'exécution d'un programme Ada.
- Doit permettre d'exécuter un même programme de façon identique sur des matériels/systèmes d'exploitation différents.
- Adapte les services du système d'exploitation aux abstractions du langage.
- Attention, une runtime peut ne pas contenir tous les éléments du langage:
  1. Le compilateur peut aider à détecter les éléments disponibles ou non.
  2. Le paquetage `System` peut aider à détecter les éléments disponibles ou non.

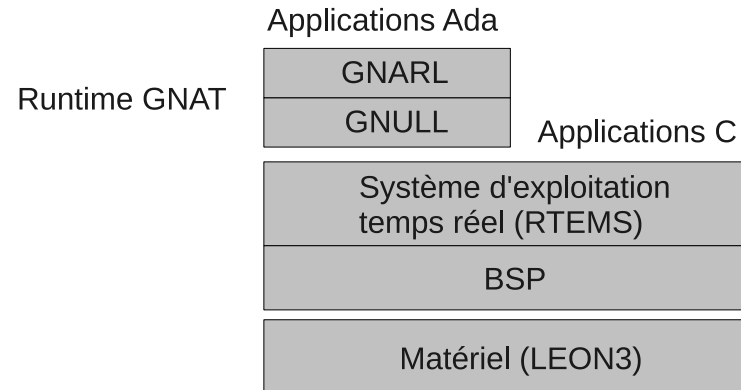
# Exemples de runtimes Ada (2)

---

- Open-Ravenscar, basé sur le système d'exploitation ORK. Compatible Ada 2005. (Universidad Politécnica de Madrid, <http://polaris.dit.upm.es/~ork/>).
- Marte OS + compilateur GNAT. (Universidad de Cantabria, <http://marte.unican.es/>).
- Runtime GNAT disponibles pour Windows, Linux, Solaris et de nombreux autres systèmes (AdaCore, <http://www.adacore.com/>).
- RTEMS operating system (OAR Corporation, <http://www.rtems.com/>).
- ...

# Exemples de runtimes Ada (3)

---



- **Runtime RTEMS :**

- RTEMS : système d'exploitation temps réel/GNU GPL pour applications C et Ada critiques de faible envergure.
- Disponible pour de nombreux BSP (dont processeur Leon : 32 bits, VHDL open-source, SMP ou AMP, compatible SPARC, applications aéronautiques/spatials).
- Compilateur GNAT/Ada 2005 (société AdaCore).
- Compilation croisée : compilation sur Linux, exécution sur Leon.

# Exemples de runtimes Ada (4)

---

- **Compilation croisée :**

1. Compilation sur Linux et génération d'un binaire SPARC :

```
#sparc-rtems4.8-gnatmake coucou
sparc-rtems4.8-gcc -c coucou.adb
sparc-rtems4.8-gnatbind coucou.ali
sparc-rtems4.8-gnatlink coucou.ali -o coucou.obj
sparc-rtems4.8-size coucou.obj
  text    data    bss    dec    hex filename
288800  13012  17824  319636  4e094 coucou.obj
sparc-rtems4.8-nm coucou.obj >coucou.num
#file coucou.obj
coucou.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
```



# Exemples de runtimes Ada (5)

---

- **Compilation croisée (suite) :**

- 2 Transfert du binaire vers la carte Leon (lien série, TCP/IP, ...)
- 3 Exécution sur une carte Leon. Logiciel `tsim` (émulateur de processeur Leon 3).

```
#tsim-leon3 coucou.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 **
coucou
Program exited normally.
tsim>
```

# Exemples de runtimes Ada (6)

---

- **Paquetage `systems.ads` pour RTEMS :**

```
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;
  ...
```

# Exemples de runtimes Ada (7)

---

- **Runtime GNAT intel/Linux :**
  - Linux/intel : système d'exploitation non temps réel mais service temps réel souple via POSIX 1003.
  - Destiné aux applications temps réel non critiques.
  - Compilateur/runtime GNAT.
  - Compatible Ada 2005 mais aussi POSIX 1003 (avec binding Ada/POSIX 1003.5, floris)
  - Pas de compilation croisée.

# Exemples de runtimes Ada (8)

---

- **Spécificités :**
  - Rappel ordonnancement Linux : priorité 0 pour `SCHED_OTHER` (temps partagé) et priorités 1 à 99 pour `SCHED_FIFO/SCHED_RR` (temps réel).
  - Les changements de priorité nécessitent les privilèges `root`, ignorés dans le cas contraire.
  - GNAT/intel/Linux traduit les priorités Ada en priorités Linux selon la politique du processus :
    1. Processus en `SCHED_OTHER` : priorités Ada ignorées.
    2. Processus en `SCHED_RR` ou `SCHED_FIFO` : priorités Ada traduites dans la plage des priorités offertes par Linux pour ces politiques.

# Exemples de runtimes Ada (9)

---

- **Paquetage System de GNAT/Linux intel:**

```
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;
```

```
    ...
```

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.

# Résumé

---

- **Pourquoi Ada** : fiabilité, abstractions temps réel, compilation séparée, portabilité (modulo les capacités/caractéristiques de la runtime).
- **Concurrence, synchronisation et communication** : tâches Ada, rendez vous et objets protégés.
- **Implanter une tâche périodique** : priorités, réveils périodiques, instant critique.
- **Être compatible avec les méthodes analytiques d'ordonnançabilité.**
- **Notions de compilation croisée et de runtime.**

# Sommaire

---

1. Généralités sur les systèmes embarqués temps réel.
2. Introduction au langage Ada 2005.
3. Concurrence.
4. Temps réel.
5. Exemples de runtimes Ada.
6. Résumé.
7. Références.



# Références

---

- [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.
- [J. 93] J. M. Rifflet. *La programmation sous UNIX*. Addison-Wesley, 3rd edition, 1993.
- [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.
- [RIF 95] J. M. Rifflet. *La communication sous UNIX : applications réparties*. Ediscience International, 2nd edition, 1995.

# Références

---

- [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.
- [TIM 00] M. Timmerman. « RTOS Market survey : preliminary result ». *Dedicated System Magazine*, (1):6–8, January 2000.
- [VAH 96] U. Vahalia. *UNIX Internals : the new frontiers*. Prentice Hall, 1996.