

Sistemas En Tiempo Real

Paquetes Incluidos

```
with Ada.Text_IO; use Ada.Text_IO;  
with pkg_ejercicio2;  
with Ada.Integer_Text_IO;
```

Paquete de impresión de texto por terminal

Paquete personalizado

Paquete de impresión de valores enteros

Las estructuras básicas de cada uno de los paquetes tienen la siguiente jerarquía.

- Una **unidad de programa** es:
 - un procedimiento
 - una función
 - un paquete

```
with Ada.Text_Io;  
use Ada.Text_Io;  
  
procedure Hola_Mundo is  
begin  
    Ada.Text_Io.Put_Line("Hola mundo");  
end Hola_Mundo;
```

Al usar un paquete podemos ahorrarnos el espacio de nombre.

Procedimiento anidado al principal

```
6 procedure Ejercicio1 is  
7     s:String := "Comenzando las prácticas de STR";  
8     numero: Natural:=0;  
9 begin  
10    Ada.Text_Io.Put("Hola Mundo!!!");  
11    Ada.Text_Io.Put_Line(s);  
12    pkg_ejercicio2.otroMensaje;  
13  
14    --Ejercicio 4a  
15    begin  
16        Put("Introduce un numero");  
17        --Recogemos un entero del usuario en la variable numero  
18        Ada.Integer_Text_IO.Get(numero);  
19  
20  
21  
22        case numero is  
23            when 1 | 2 | 12 => Put_Line("Invierno");  
24            when 3 | 4 | 5 => Put_Line("Primavera");  
25            when 6 | 7 | 8 => Put_Line("Verano");  
26            when 9 | 10 | 11 => Put_Line("Otoño");  
27            when others => Put_Line("Mes incorrecto");  
28        end case;  
29  
30  
31        exception  
32            when CONSTRAINT_ERROR =>  
33                Put("El numero debe ser > 0");  
34                New_Line;  
35  
36    end;  
37  
38    --Ponemos aqui el fin del programa para que no se repita  
39    Put_Line("FIN DEL PROGRAMA");  
40  
41 end Ejercicio1;
```

Variables globales que podemos usar a lo largo del programa

Escribimos un mensaje por pantalla llamada a una función de un paquete exterior

Excepción personalizada por el usuario

Paquetes personalizados por el usuario:

■ Cada **unidad de programa** (excepto la del programa principal) se divide en dos partes:

- una **especificación**, que define su interfaz con el "mundo exterior" (*.ads) literalmente el .h del C++
- un **cuerpo**, que contiene los detalles de implementación (*.adb) literalmente el .c de C++

Especificación del paquete

```
1 with Ada.Text_IO, Ada.Float_Text_IO;
2 package pkg_ejercicio2 is
3   --Esto ya era un enumerado creo
4   type TdiasSemana is (Lunes, Martes, Miercoles,
5                         Jueves, Viernes,
6                         Sabado, Domingo);
7   --Transformamos el TdiasSemana en un enumerado legible para poder hacer input output
8   package enumerado_TdiasSemana is new Ada.Text_IO.Enumeration_IO(Enum => TdiasSemana);
9
10  numAlumnos: Integer := 19;
11  procedure otroMensaje;
12  function getNotaMedia return Float;
13
14  private
15    notaMedia: Float:=6.12;
16 end pkg_ejercicio2;
```

Variables legibles y enumeradas:

--Esto ya era un enumerado creo

```
type TdiasSemana is (Lunes, Martes, Miercoles,
                      Jueves, Viernes,
                      Sabado, Domingo);
```

En el enumerado TdiasSemana definimos los valores que va a contener, pero este no podrá ser legible para ello tendremos que generar un paquete legible de enumerados

--Transformamos el TdiasSemana en un enumerado legible para poder hacer input output
package enumerado_TdiasSemana is new Ada.Text_IO.Enumeration_IO(Enum => TdiasSemana);

Ejemplo de como recorrer el enumerado y imprimir sus valores.

```
--Bucle donde la variable dia adquiere valores enteros
for dia in pkg_ejercicio2.TdiasSemana loop
  --Imprimimo el valor de la posicion de dia
  pkg_ejercicio2.enumerado_TdiasSemana.Put(dia);
  New_Line;
end loop;
```

Comprobamos si un valor esta dentro de un enumerado.

```
--Recogemos un valor del usuario en este caso un string
Put(Item => "Introduce un dia cualquiera");
New_Line;

--Buscamos dentro del enumerado el item sino es parte del salta excepcion
pkg_ejercicio2.enumerado_TdiasSemana.Get(diaS);
```

Recorremos el no legible pero imprimimos los valores con el Put del legible pasandole el valor que queremos imprimir

Si no esta salta la excepción

```
--Sacamos la posicion del valor introducido
pos:= pkg_ejercicio2.TdiasSemana'Pos(diaS);
--Ada.Integer_Text_IO.Put(pos);
--Obtenemos el valor de la posicion dentro del tipo
valor:= pkg_ejercicio2.TdiasSemana'Val(pos);
```

Cuerpo del paquete

```
1 with Ada.Text_IO;
2 use Ada.Text_IO;
3
4 package body pkg_ejercicio2 is
5   procedure otroMensaje is
6   begin
7     Put_Line("Vamos a iniciarnos en el lenguaje Ada");
8   end otroMensaje;
9
10  function getNotaMedia return Float is
11  begin
12    return notaMedia;
13  end getNotaMedia;
14
15 end pkg_ejercicio2;
```

Variables y procedimientos publicos

Variables y procedimientos privados.

Subtypes: los Subtypes son valores de un tipo concreto (Integer, Float, etc.) que cumplen una restricción en concreto

Ejemplo:

```
subtype Negative is Integer range Integer'First..-1;
subtype Uppercase is Character range 'A'..'Z';
subtype Probability is float range 0.0..1.0;
```

Bloques y Manejo de Excepciones:

```
begin
  ...
  exception
    when Error1 | Error2 => instrucciones a ejecutar si ocurre la excepción Error1 o la Error2;
    when Error3 => instrucciones a ejecutar si ocurre la excepción Error3;
  end;
```

Mi excepción

```
exception
  when CONSTRAINT_ERROR =>
    Put("El numero debe ser > 0");
    New_Line;
```

Generación de números aleatorios:

Importamos el paquete para generar números aleatorios

```
with Ada.Numerics.Discrete_Random; --paquete random generico
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Text_IO; use Ada.Text_IO;
```

Dentro del procedimiento realizar una serie de pasos

```
procedure ejercicio6 is
  subtype T_Dígito is Integer range 0..9; ①
  package Pkg_DígitoAleatorio is new Ada.Numerics.Discrete_Random (T_Dígito) ②
  generator_dígito : pkg_DígitoAleatorio.Generator; -- declarar generador de
  --valores aleatorios tipo T_Dígito
  dígito: T_Dígito; ④
  ③ Instanciamos el generador
  ④ Declaramos la variable donde
  almacenaremos el dígito.
```

- ① Instanciamos un subtipo de tipo entero que tendrá el rango de generación de aleatorios
- ② Instanciamos el paquete con el rango

Procedimiento que imprime números aleatorios

```
begin
  ⑤
  pkg_DígitoAleatorio.Reset (Generador_Dígito); -- Inicializa generador números aleatorios
  loop
    dígito:= Pkg_DígitoAleatorio.Random(generator_dígito);
    --Se esta utilizando el paquete de Integer_Text_IO ya que se le pasa un entero
    Put(dígito);
    Skip_Line;
  end loop;
end ejercicio6;
```

- ⑤ Presetamos la semilla generadora
- ⑥ Generamos el número aleatorio y luego lo imprimimos.

En el caso de que queramos generar números aleatorios pero de tipo Float.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Numerics.Float_Random;
with Ada.Float_Text_IO;

procedure numeroAleatorios is
  subtype T_Float is Float range 0.0..1.0;
  generador_float : Ada.Numerics.Float_Random.Generator; -- declarar generador de valores de tipo Float
  dígito : T_Float;
begin
  Ada.Numerics.Float_Random.Reset(generador_float); -- Inicializar generador de nº aleatorios
  for i in 0..5 loop
    dígito := Ada.Numerics.Float_Random.Random(generador_float); -- generar nº aleatorio
    Ada.Float_Text_IO.Put(dígito, Aft => 4, Exp => 0);--Aft Exp cantidad de decimales que queremos
    Ada.Text_IO.Skip_Line;
  end loop;
end numeroAleatorios;
```

Procedimientos y funciones genéricas:

```
-- Especificación  
generic  
    type TData is private;  
    procedure Intercambiar (d1: in out TData; d2: in out TData);  
-- Instanciación  
Procedure Intercambiar_Entero is new Intercambiar(TData=>Integer);  
Procedure Intercambiar_Caracter is new Intercambiar(TData => Character);
```

Son como el paso por referencia si pusieramos solo out tendríamos que pasársela una variable sin inicializar para asignar un valor.

Ejemplo de paquetes genéricos creados por el usuario:

```
generic  
    type TData is private; -- parámetros genéricos  
package Store is  
    type TBuffer is limited private;  
    procedure Give (d: TData; b: in out TBuffer);  
    procedure Take (d: out TData; b: in out TBuffer);  
  
    private -- declaraciones privadas  
        size : constant := 80;  
        type TVector is array (1..size) of TData;  
        subtype TLon is integer range 0..size;  
  
        type TBuffer is  
            record  
                vector: TVector;  
                lon : TLon:= 0;  
            end record;  
    end Store;  
  
    package body Store is  
        -- implementación  
    end Store;
```

Este buffer es capaz de almacenar datos de cualquier tipo

Como usar el buffer especificado:

-- Utilización

```
type TDate is  
    record  
        Day: Integer range 1..31;  
        Month: Integer range 1..12;  
        Year: Integer range 1066..2066;  
    end record; Nombre que le ponemos  
  
package Date_Store is new Store(TData => TDate);  
package Int_Store is new Store(TData => Integer);  
Variables donde almacenamos.  
BufferI : Int_Store.TBuffer;  
BufferD : Date_Store.TBuffer;  
  
i : integer;  
date : TDate  
  
... Usos  
Int_Store.Give(i, BufferI);  
Date_Store.Take(date, BufferD);
```

Tipo de datos que guardamos.

Paquetes predefinidos:

- Operaciones con caracteres y cadenas:
 - Ada.Characters
 - Ada.String
- Cálculo numérico:
 - Ada.Numerics
 - Ada.numerics.Generic_elementary_functions .
- Entrada y Salida:
 - Ada.text_io
 - Ada.integer_text_io
 - Ada.float_text_io
 - Gnat.io

Declaración de tareas:

Las tareas se pueden declarar dentro de cualquier bloque declarativo.

- Un subprograma
- Un bloque
- Un paquete
- el cuerpo de otra tarea.

Están formados por una especificación y un cuerpo.

Tipo tarea definido por usuario

```
task type Tipo_A;  
task type Tipo_B;  
  
A : Tipo_A;  
B : Tipo_B;
```

Cuando tengamos que lanzar varias procesos de un mismo comportamiento.



Especificación

```
task type T_tarea (X : tipo_parametro) is  
    -- declaración de interfaz con otras tareas  
private  -- opcional  
end T_Tarea;
```

Tipo anónimo

```
task A;  
task B;
```

```
task type T_tarea;  
    -- si no tiene interfaz con  
    -- otras tareas
```

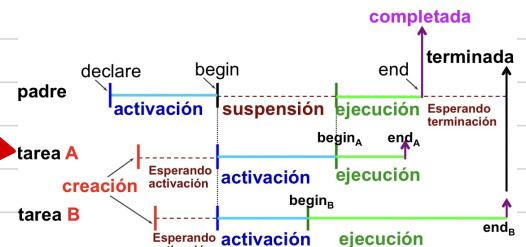
Cuerpo de la tarea:

```
task body T_Tarea is  
    -- declaraciones locales  
begin  
    -- implementación del comportamiento  
    -- de la tarea  
end T_Tarea;
```

declare

```
...  
A: T_tarea;  
B: T_tarea;  
...  
begin  
...  
end;
```

Fases de una Tarea



Creación de la tarea:

```

Procedure Ejemplo is
    task type T_tarea;
    type Ptr_T_tarea is access T_Tarea;

    task body T_tarea is
        begin
        ...
    end;

    Tarea_E : T_tarea; -- CREACIÓN DE TAREA ESTÁTICA
    Tarea_D : Ptr_T_Tarea; -- declaración de puntero a tipo tarea
    ...
    Tarea_D := new T_Tarea; -- CREACIÓN DE TAREA DINÁMICA
end Ejemplo;

```

Crea la tarea y la activa

Tareas Estáticas:

Se activan justo antes de empezar la ejecución de la unidad

Tareas dinámicas:

Se activan al ejecutarse el operador new.

Ejemplo de Creación y activación

```

Procedure Ejemplo is
    task type T_tarea;
    type Ptr_T_tarea is access T_Tarea;

    task body T_tarea is
        begin
        ...
    end;

    R : T_tarea; -- CREACIÓN de la tarea R
    P : Ptr_T_Tarea;
    Q : Ptr_T_Tarea := new T_Tarea; -- CREACIÓN Y ACTIVACIÓN
                                    -- de la tarea Q.all
    Begin -- ACTIVACIÓN de la tarea R
        P := new T_Tarea; -- CREACIÓN Y ACTIVACIÓN de P.all
        Q := new T_Tarea; -- CREACIÓN Y ACTIVACIÓN de otra tarea Q.all
                           -- La primera tarea Q sigue ejecutándose,
                           -- pero pasa a ser anónima
    end Ejemplo;

```

Tarea que se inicializa justo antes del begin

Declara un puntero a una tarea que se inicializará en el begin

Tareas dinámicas que devuelven un puntero que se inserta en la variable

Práctica 2 genera aviones

Especificación

```
package GeneraAviones is
    TASK TareaGeneraAviones;
    -- Tipo tarea encargada del comportamiento de un avion
    TASK TYPE T_TareaAvion(ptr_avion : Ptr_T_RecordAvion);
    --Puntero al avion para poder pasarselos datos
    type Ptr_TareaAvion is access T_TareaAvion;
end GeneraAviones;
```

A las tareas solo les podemos pasar punteros, los cuales nos dan acceso a esa tarea lo que nos permite modificar su comportamiento, en este caso le pasamos un puntero al record que contiene los datos

Tarea de avión
a la que le
pasaremos la info
de un avión y se
encargará de su lógica.

Contendrá toda la
información de un
avión

Información del
avión que guardamos
en el puntero

Creamos la nueva
tarea de avión la
cuál representará un
avión y le pasamos
la información
guardada en el
puntero

TareaGeneraAviones

```
1 package body GeneraAviones is
2
3     task body TareaGeneraAviones is
4
5         --Puntero al avion
6         tarea_avion : Ptr_TareaAvion;
7         --Puntero a los datos del avion
8         ptr_avion : Ptr_T_RecordAvion;
9
10        --Rango de aparición aleatorio
11        package Pkg_generadorRetardo is new Ada.Numerics.Discrete_Random (T_RetardoAparicionAviones);
12        generador_retardo: Pkg_generadorRetardo.Generator;
13        --retardo : T_RetardoAparicionAviones;
14
15
16        --Rango de color aleatorio
17        package Pkg_generadorColor is new Ada.Numerics.Discrete_Random (T_ColorAparicionAvion);
18        generador_color: Pkg_generadorColor.Generator;
19        --retardo : T_RetardoAparicionAviones;
20
21
22        begin
23            --Resetear las semillas
24            Pkg_generadorRetardo.Reset(generador_retardo);
25            Pkg_generadorColor.Reset(generador_color);
26
27
28            for id in T_IdAvion loop
29                for aereovia in T_Rango_AereoVia'First..T_Rango_AereoVia'Last - 2 loop
30
31                -- inicializar datos de un nuevo avion
32                ptr_avion := new T_RecordAvion;
33
34
35                --Datos del avion
36                ptr_avion.id := id;
37                ptr_avion.velocidad.x := VELOCIDAD_VUELO;
38                ptr_avion.velocidad.y := 0;
39                ptr_avion.aereovia := aereovia;
40                ptr_avion.tren_terriza := False;
41                ptr_avion.aereovia_inicial := aereovia;
42                ptr_avion.pista := SIN_PISTA;
43                ptr_avion.color := Pkg_generadorColor.random(generador_color);
44                ptr_avion.pos := Pos_Inicio(aereovia);
45
46                --Cambia la dirección de vuelo del avion
47                if ptr_avion.aereovia / 2 = 0 then
48                    ptr_avion.velocidad.X := VELOCIDAD_VUELO;
49                else
50                    ptr_avion.velocidad.X := -VELOCIDAD_VUELO;
51                end if;
52
53
54                -- Crear una tarea para el comportamiento del avion
55                tarea_avion := NEW T_TareaAvion(ptr_avion);
56
57                delay(Duration(Pkg_generadorRetardo.random(generador_retardo)));
58                --Anadir delay generando numero aleatorio
59
60
61            end loop;
62            end loop;
63
64
65
66
67        end TareaGeneraAviones;
```

Generamos los nuevos
paquetes con el
rango que deseamos
ademas de un generador
de números

Reseteamos las
semillas de los
generadores

Le metemos un
delay al bucle

TEMA 2

- Sistema en tiempo real: sistema que interacciona repetidamente con su entorno físico, responde a estímulos de dicho entorno en un tiempo determinado.
 - Por lo general parte de un sistema embebido
 - Tiene recursos limitados.
 - Dispositivo de entrada y salida especial
 - Inaccesibles desde el exterior
 - Aplicación se ejecuta desde Rom.

Principales Características:

- Concurrencia:
 - * Modelar el paralelismo en el mundo real
 - * Inherente mente concurrentes.
- Dependencia del tiempo:
 - * Tareas terminan antes de la deadline
 - * No determinista
 - * Pueden no cumplir restricciones temporales.
 - * Predecible
- Fiabilidad y seguridad

Otras Características:

- Interacciona con dispositivos físicos
- Gran tamaño y complejidad
- Cálculo con números reales
- Datos volátiles y Pruebas.

Tipos de STR

- Según Criticidad
 - * Críticos
 - * Acríticos
 - * Firme
- Según Activación:
 - * Dirigidos por eventos
 - * Dirigidos por tiempo

Críticos: todas las acciones deben ocurrir en un plazo especificado, las respuestas tardías pueden tener consecuencias fatales

Acríticos: Se pueden perder plazos y el valor de la respuesta decrece con el tiempo.

Firme: Se pueden perder plazos y las respuestas tardías no tienen valor

Dirigidos por tiempo: Inicio en instantes determinados, mecanismo básico de reloj

Dirigidos por eventos: Se produce un suceso de cambio de estado, mecanismo de interrupción

Esquemas de activación:

- **Periódica**: Se ejecuta regularmente con un periodo bien definido
- **Aperiódica**: Se ejecuta de manera irregular en respuesta a un suceso del entorno
- **Esporádica**. Se exige una separación mínima de dos sucesos consecutivos.

Arquitecturas de software posibles:

- **Ejecutivo cíclico**: Único programa secuencial con un bucle de control, ignorando la evidente concurrencia
- **Sistemas Operativos de tiempo real**: Se escribe en un lenguaje de programación secuencial
- **Lenguaje de programación de tiempo Real**: un único programa concurrente

Problema de espera ocupada: el programa consume una gran proporción de su tiempo en un bucle ocupado, comprobando si los dispositivos de entrada están disponibles. Son ineficientes.

Criterios de Diseño:

Características	Lenguaje de TR	Lenguaje Ada
Concurrencia	Procesos simultáneos	Tareas, objetos protegidos, citas extendidas
Dependencia del tiempo	Especificación y análisis	Librería de paquetes: Calendar y Real_Time
Fiabilidad y seguridad	Legibilidad y mecanismos de recuperación de errores	Fuertemente tipado, manejadores de excepciones
Interacción con el hardware	Control de interrupciones y drivers	Manejador de interrupciones
Tamaño y complejidad	Modularidad y portabilidad	Paquetes y unidades genéricas

TEMA 3

Requerimientos temporales en STR.

- Interacción con el tiempo:
 - Medir el paso de tiempo con relojes
 - Retrasar la ejecución de las tareas
 - programación de timeouts
 - Ejecutar acciones en determinados instantes
- Representación de requerimientos temporales
 - Periodos de activación
 - Plazos de ejecución deadlines
- Satisfacción de requerimientos temporales
 - Scheduling

Facilidades de control de tiempo Real :

- Especificar los tiempos en los que las acciones tienen que ejecutarse y completarse.
- Responder a las situaciones en las que no se pueden atender todos los plazos y los requerimientos temporales cambian.

Medir el paso del tiempo:

Mediante acceso directo al marco temporal del entorno

Mediante el uso de un reloj hardware interno que proporcione una aproximación adecuada del paso del tiempo en el entorno.

Acceso a relojes en Ada

Paquetes	Ada.Calendar	Ada.Real_Time
Tipo de datos para instantes de tiempo absoluto	tipo Time	tipo Time
Tipo de datos para intervalos de tiempo	tipo Duration	tipo Time_Span
Función Clock	Devuelve un valor que combina la fecha y la hora actual	Devuelve un valor monótono creciente que representa el tiempo transcurrido desde un instante inicial prefijado

Ada Calendar:

```

package Ada.Calendar is
    type Time is private;
    subtype Year_Number is Integer range 1901..2099;
    subtype Month_Number is Integer range 1..12;
    subtype Day_Number is Integer range 1..31;
    subtype Day_Duration is Duration range 0.0..86400.0;
    function Clock return Time;
    function Year(Date:Time) return Year_Number;
    function Month(Date:Time) return Month_Number;
    function Day(Date:Time) return Day_Number;
    function Seconds(Date:Time) return Day_Duration;
    procedure Split(Date:in Time; Year:out Year_Number;
                    Month:out Month_Number; Day:out Day_Number;
                    Seconds:out Day_Duration);
    function Time_Of(Year:Year_Number; Month:Month_Number;
                    Day:Day_Number; Seconds:Day_Duration := 0.0) return Time;
    function "+"(Left:Time; Right:Duration) return Time;
    function "+"(Left:Duration; Right:Time) return Time;
    function "-"(Left:Time; Right:Duration) return Time;
    function "-"(Left:Time; Right:Time) return Duration;
    function "<"(Left,Right:Time) return Boolean;
    function "<="(Left,Right:Time) return Boolean;
    function ">"(Left,Right:Time) return Boolean;
    function ">="(Left,Right:Time) return Boolean;

    Time_Error:exception;
        -- Time_Error may be raised by Time_Of,
        -- Split, Year, "+" and "-"

private
    -- implementation-dependent
end Ada.Calendar;

```

Ada Real_Time

```

package Ada.Real_Time is
    type Time is private;
    Time_First: constant Time;
    Time_Last: constant Time;
    Time_Unit: constant := implementation_defined_real_number;

    type Time_Span is private;
    Time_Span_First: constant Time_Span;
    Time_Span_Last: constant Time_Span;
    Time_Span_Zero: constant Time_Span;
    Time_Span_Unit: constant Time_Span;
    Tick: constant Time_Span;

    function Clock return Time;
    function "+" (Left: Time; Right: Time_Span) return Time;
    function "+" (Left: Time_Span; Right: Time) return Time;
    -- similarly for "-", "<", etc.
    function To_Duration(TS: Time_Span) return Duration;
    function To_Time_Span(D: Duration) return Time_Span;

    function Nanoseconds (NS: Integer) return Time_Span;
    function Microseconds(US: Integer) return Time_Span;
    function Milliseconds(MS: Integer) return Time_Span;

    type Seconds_Count is range implementation-defined;
    procedure Split(T : in Time; SC: out Seconds_Count;
                    TS : out Time_Span);
    function Time_Of(SC: Seconds_Count;
                    TS: Time_Span) return Time;

private
    -- not specified by the language
end Ada.Real_Time;

```

Time: proporciona instantes de tiempo absolutos
Duration: proporciona intervalos de tiempo relativos

```

declare
    with Ada.Calendar; use Ada.Calendar;
    Inicio, Fin : Time;
    Duracion : Duration;
begin
    Inicio := Clock;
    ... -- secuencia de instrucciones
    Fin := Clock;
    Duracion := Fin - Inicio;
end;

```

Time: representa el tiempo transcurrido desde el comienzo de la ejecución del programa o de un determinado instante de la época.

Time_Span: representa intervalos de tiempo

```

declare
    with Ada.Real_Time; use Ada.Real_Time;
    Comienzo, Fin : Time;
    Duracion : Time_Span := To_Time_Span(1.7);
    -- o bien Time_Span := Milliseconds(1700);
begin
    Comienzo := Clock;
    ... -- secuencia de instrucciones
    Fin := Clock;
    if Fin - Comienzo > Duracion then
        raise Tiempo_Excedido; -- excepción definida por usuario
    end if;
end;

```

Concepto y tipos de retardo:

- Suspender la ejecución de la tarea durante cierto tiempo.
- Tipos de retardos:
 - Relativo: la ejecución se suspende durante un intervalo de tiempo relativo al instante actual.
 - Absoluto: la ejecución se suspende hasta que se llegue a un instante determinado de tiempo.

Retrasar una tarea

```
Comienzo := Clock; -- de Ada.Calendar
loop
    exit when (Clock - Comienzo) > 10.0;
end loop;
```

→ Esto es una espera ociosa y debemos evitarlas

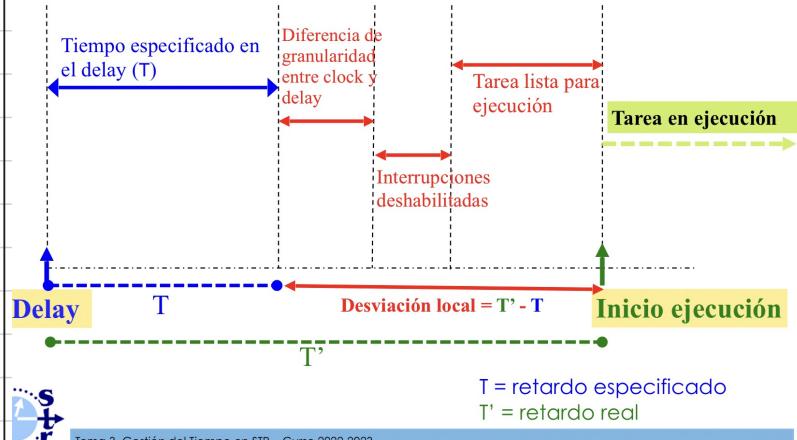
Retardos relativos en Ada:

delay (T);

Suspende la ejecución de la tarea que la invoca durante el intervalo de tiempo que indica el valor T

→ T: Duration (Se mide en segundos)

Ejecución de un retardo:



Retardos absolutos en Ada:

delay until (T);

Suspende la ejecución de la tarea que la invoca hasta que el valor `clock` sea igual al especificado por T

Tarea periódica en Ada

Desviación Acumulativa: Acumula delay lo que puede llevar a errores

```
task T;  
task body T is  
begin  
loop  
  Accion;  
  delay 5.0;  
end loop;  
end T;
```

Desviación local: Esta no acumula errores, se ejecuta con una media

```
task body T is  
  periodo: constant Duration := 5.0;  
  Siguiente : Time;  
begin  
  Siguiente := Clock + periodo;  
loop  
  Accion;  
  delay until Siguiente;  
  Siguiente := Siguiente + periodo;  
end loop;  
end T;
```

Cuidado: Si aquí ponemos clock en vez de siguiente estaría completamente mal ya que acumularíamos el error.

Programación de timeouts: limitar el tiempo durante el cual una tarea está lista para esperar una comunicación.

- Acceso a memoria compartida
- Pase de mensajes entre tareas

Limitar el tiempo de ejecución de una acción:

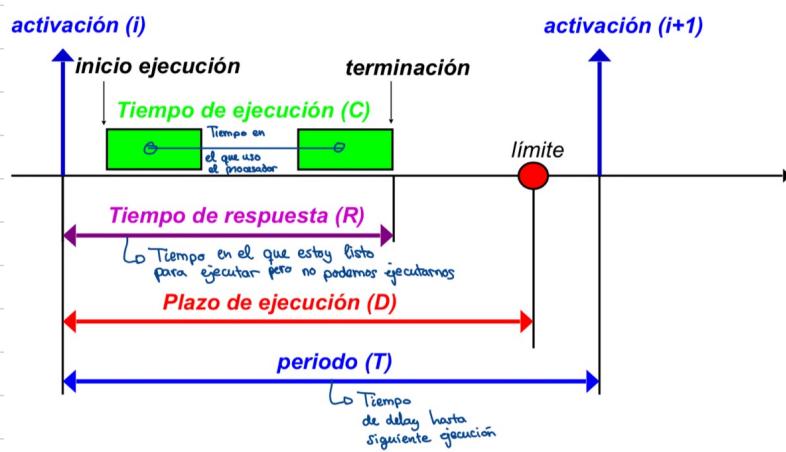
- detección de pérdidas de plazo
- Aplicación al computo impreciso.

```
begin  
  -- parte obligatoria  
  Escribe(...); -- actualizamos el resultado inicial  
  select  
    delay until Fin_Plazo;  
  then abort  
    loop  
      -- mejorar el resultado  
      Escribe(...);  
    end loop;  
  end select;  
end;
```

Tiempo que tiene la tarea para ejecutarse.

Esta acción se ejecuta hasta que acaba el tiempo

Ejecución de una tarea de tiempo Real



Especificación:

```
package GeneraCoches is
```

Type Tcolor is (Rojo, Rosa, Verde); Enumerado para los colores

Subtype espera is Integer range 1000...10000; Range para las esperas

Task generaCosas; Task padre encargada de generar a los hijos

type datosCosa is record

 Color : Tcolor

 espera : espera

end record;

Type **p-datosCosa** is access datosCosa; Puntero para entrar al record

} Record para almacenar los datos

Task type cosa (d-cosa : p-datosCosa); Task hija encargada de almacenar los valores

y imprimir

type p-cosa is access cosa; Puntero de acceso a la Task

end GeneraCosas;

Cuerpo del paquete:

```
package GeneraCosas is
```

Task generaCosas is

 Periodo : Time; → Variable para el clock.

 ptr-datos : p-datosCosa; → puntero para pasar datos a la tarea

 ptr-Cosa : p-cosa, → puntero para crear la tarea dinámicamente.

{ package Pkg-generaEsperas is new Ada.Numerics.Discrete_Random (espera);

{ generate_Random Espera : Pkg-generaEsperas.generator

{ package Pkg-generaColores is new Ada.numerics.Discrete_Random (Tcolor);

{ generate_Random Color : Pkg-generaColores.generator;

begin

 pkg-generaEsperas.reset (generate_Random Espera); } Resetteamos las semillas

 pkg-generaColores.reset (generate_Random Color); }

 Periodo : Clock → Iniciamos el reloj.

loop

 ptr-datos := new datosCosa;

 ptr-datos.color = Pkg-generaColores.Random (generate_Random Color);

 ptr-datos.espera = Pkg-generaEsperas.Random (generate_Random Espera);

Hacemos la suma { periodo := periodo + Milliseconds (ptr-datos.espera);
del tiempo y { delay until periodo;
realizamos delay ptr-cosa = new Cosa (ptr-datos), → creamos el puntero y la tarea

end loop

end generaCosas;

Task Body Cosa is

 dato : datosCosa;

begin

 dato := d-cosa.all;

 Put-Line ("Tiempo Espera de Cosa" & dato.espera'Image);

 Put-Line ("Color de Cosa" & dato.color'Image);

end Cosa;

end GeneraCosas;

