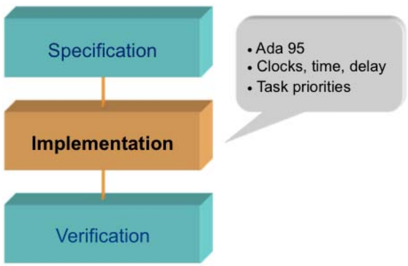
CHALMERS

Real-Time Systems



CHALMERS

Ada 95 Reference Manual (ARM)

Ada 95 consists of a core language and a set of annex containing extensions for special applications.

An Ada 95 implementation must support the entire *core language*, but can choose to support an arbitrary combination of annex.

An annex may define new packages, attributes and pragma, but may not introduce new syntax or change semantics of the *core language*.

CHALMERS

Ada 95 Reference Manual (ARM)

The following parts of ARM are dealt with in this course:

Section 9:	Tasks and Synchronization
Section 13:	Representation Issues
Annex C:	Systems Programming
Annex D:	Real-Time Systems

In addition, the following parts of ARM are interesting:

Annex E:	Distributed Systems
Annex F:	Information Systems
Annex G:	Numerics
Annex H:	Safety and Security

CHALMERS

Clocks and time in Ada 95

To construct a real-time system, the chosen programming language must support a concept of time that can be used for modeling the system's time constraints.

In Ada 95, time is represented as system clocks, that can be read in order to report current time.

Ada 95 has two different time packages that each defines a system clock:

- Ada.Calendar: compulsory package (Section 9.6) with a clock that represents calendar time with "satisfactory" resolution.
- Ada.Real_Time: annex package (Annex D.8) with a clock that represents physical (monotonic) time with high resolution.

CHALMERS

Calendar time in Ada 95

Ada.Calendar defines a data type **Time** that represents calendar time (date + seconds since midnight) with a resolution of at least 20 ms. Values of this type can be converted to year, month, day and seconds.

Calendar time is normally monotonic (non-decreasing), but can be adjusted (forwards/backwards) as a consequence of e.g. daylight savings time or other time adjustments.

The current value of the calendar time can be read by calling the function **Ada.Calendar.Clock**.

A (calendar) time interval (i.e. the difference between two time instants) is represented by the data type **Duration**.

CHALMERS

Real time in Ada 95

Ada.Real_Time defines a data type **Time** that represents real time (physical time) with a resolution of at least 1 ms. Values of this type cannot be converted to calender data.

Real time is strictly monotonic (cannot be adjusted backwards) and measured in elapsed time units since an epoch. Time unit and epoch are both implementation dependent.

The current value of the real time can be read by calling the function **Ada.Real_Time.Clock**.

A (real) time interval (i.e. the difference between two time instants) is represented by the data type **Time_Span**.

Although same names are used for types & functions, **Ada.Calendar** and **Ada.Real_Time** can coexist in the same program.

CHALMERS

Example: control of execution time
(with Ada.Calendar)

```
with Ada.Calendar;
use Ada.Calendar;
package body Controller is
  task body Temp_Controller is
    ...        -- declaration of variables
    Start, Finish : Time;
    Interval : Duration := 1.7;
    Overrun_Error : exception;
  begin
    loop
      Start := Clock;
      ...              -- statements in Temp_Controller;
      Finish := Clock;
      if Finish - Start > Interval then
        raise Overrun_Error;
      end if;
    end loop;
  exception
    when Overrun_Error =>
      -- program code for error handling
  end Temp_Controller;
end Controller;
```

CHALMERS

Example: control of execution time
(with Ada.Real_Time)

```
with Ada.Real_Time;
use Ada.Real_Time;
package body Controller is
  task body Temp_Controller is
    ...        -- declaration of variables
    Start, Finish : Time;
    Interval : Time_Span := To_Time_Span(1.7);
    Overrun_Error : exception;
  begin
    loop
      Start := Clock;
      ...              -- statements in Temp_Controller;
      Finish := Clock;
      if Finish - Start > Interval then
        raise Overrun_Error;
      end if;
    end loop;
  exception
    when Overrun_Error =>
      -- program code for error handling
  end Temp_Controller;
end Controller;
```

Time constants have type **Duration** as default.

Conversion of time intervals is found in **Ada.Real_Time**.

CHALMERS

Time delays

How can the execution of a task be delayed in Ada?

- Use the (relative) **delay** statement:

```
delay 10.0;      -- wait for 10 seconds
```

- While the task is delayed in the **delay** statement, other tasks (if such exist) may execute.
- The **delay** statement guarantees that the delay will be at least the indicated number of seconds (which should be of type **Duration**).
- The actual delay could be longer because the delayed task may have to wait for other tasks to complete their execution.

CHALMERS

Periodic activities

Example: Execute a task periodically every 5th second.

```
package body Periodic_Action is
  task body T is
    Interval : constant Duration := 5.0;
  begin
    loop
      Action;
      delay Interval;
    end loop;
  end T;
end Periodic_Action;
```

This solution gives rise to a systematic time skew

- The code for **Action** takes a certain time Δ_{action}
- The code for administrating the loop construct takes a certain time Δ_{loop}

⇒ The minimum interval between two executions of **Action** is:
$5 + \Delta_{action} + \Delta_{loop}$ seconds.

CHALMERS

Periodic activities

How can systematic time skew be avoided in Ada?

- Use the (absolute) **delay** statement:

```
delay until Later;     -- wait until clock becomes Later
```

- The absolute **delay** statement guarantees that the continued execution is delayed until the given time instant at the earliest.
- The given time instant can be of arbitrary time type (i.e. from **Ada.Calendar** as well as from **Ada.Real_Time**).

CHALMERS

Periodic activities

```
package body Periodic_Action is
  task body T is
    Interval : constant Duration := 5.0;
    Next_Time : Time;
  begin
    Next_Time := Clock + Interval;
    loop
      Action;
      delay until Next_Time;
      Next_Time := Next_Time + Interval;
    end loop;
  end T;
end Periodic_Action;
```

This solution does not eliminate local time skew

- Other tasks with same or higher priority may interfere so that the task cannot begin its execution at the desired time instant
- Local time skew may cause the start time within the current time interval to vary between different executions of the same task.
- Local time skew can be avoided by using suitable scheduling algorithms or be determined with the aid of special analysis methods.

CHALMERS

Example: a simple control system

Problem: Write a procedure `Periodic_Controller` for the control system introduced in an earlier lecture.

- Task `Temp_Controller` should use an iteration period of **70 ms**.
- Task `Pressure_Controller` should use iteration period **30 ms**.
- Printing to the display should take place without the server task.
- Use package **Ada.Real_Time** to model physical time.

We solve this on the blackboard!

CHALMERS

Task priorities in Ada 95

To be able to guarantee and analyze the behavior of a real-time system, the programming language and run-time system must have support for task priorities.

Task priorities are used for selecting which task that should be executed if multiple tasks contend over the processing resource (the CPU).

The priority of a task can be given in two different ways:

Static priorities: based on task characteristics that are known before the system is running, e.g., iteration frequency or deadline.

Dynamic priorities: based on task characteristics that are derived at certain times while the system is running, e.g., remaining execution time or remaining time to deadline.

CHALMERS

Task priorities in Ada 95

Task priorities are of data type **Any_Priority** which is declared in package System (see Section 13.7 in ARM).

Priorities are a subtype of **Integer** and are given as values in the range

```
Any_Priority'First .. Any_Priority'Last
```

The range of the priority values is implementation dependent (not defined in the language):

```
subtype Any_Priority is Integer range implementation-defined;
```

CHALMERS

Task priorities in Ada 95

Depending of the type of task, two types of priorities are used (both of which are subtypes of **Any_Priority**):

Normal tasks use priorities av data type **Priority**:

```
subtype Priority is Any_Priority
   range Any_Priority'First .. implementation-defined;
```

Interrupt handlers and protected objects use priorities of data type **Interrupt_Priority**:

```
subtype Interrupt_Priority is Any_Priority
   range Priority'Last+1 .. Any_Priority'Last;
```

CHALMERS

Static task priorities in Ada 95

In the Ada 95 *core language* there is only support for static task priorities.

The static (base) priority of a task is expressed using the pragma **Priority**, which should be located in the specification of the task.

```
task P1 is
  pragma Priority(5);
  entry E1(X : in Objekt);
  entry E2(Y : out Objekt);
end P1;
```

The parameter to the pragma is of data type **Priority**.

CHALMERS

Static task priorities in Ada 95

In the absence of a priority pragma, a task inherits the priority of its parent task.

If no priority is given in its ancestors, the task is assigned the priority **Default_Priority** (found in package **System**):

```
Default_Priority : constant Priority :=
   (Priority'First + Priority'Last)/2;
```

For the main program, which is executed by a predefined (non-declared) task, the priority is given directly in the main procedure because it lacks a specification part.

If no priority is given for the main program, it is assigned the priority **Default_Priority**.

CHALMERS

Dynamic task priorities i Ada 95

Annex D (Real-Time Systems) provides support for dynamic priorities via package **Ada.Dynamic_Priorities**:

```
package Ada.Dynamic_Priorities is
  procedure Set_Priority(...);
  function Get_Priority(...) return Any_Priority;
end Ada.Dynamic_Priorities;
```

By means of this package, the priority of a task can be read and modified while the system is running.

CHALMERS

Priorities and shared objects

When task priorities are used to introduce determinism and analyzability to the system, this must also encompass the handling of protected objects.

In order to verify the system, an upper bound of each task's blocking time must be possible to derive.

Such derivation is relatively simple as long as a task can only be blocked by tasks with higher priority.

The analysis becomes much more difficult when protected objects are used, as a task can then also be blocked by tasks with lower priority that does not use the object.

One such example is when priority inversion occurs.

CHALMERS

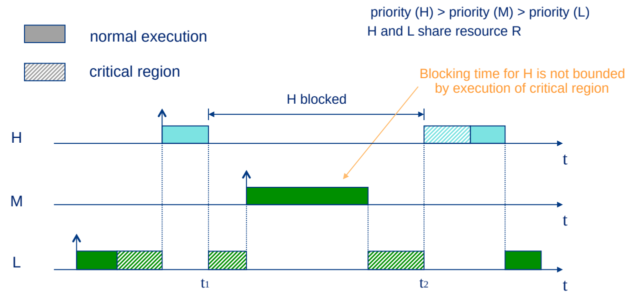
Priority inversion

Assume three tasks H, M and L (decreasing priorities) where H and L share a protected object.

1. Assume that task L with lowest priority requests and acquires a protected object (critical region).
2. Task H, which has highest priority, then starts and requests the protected object. As only one task at a time can execute code in a protected object, H must wait until L releases the object.
3. Task M, which has medium priority, preempts task L according to the rules and then starts its execution.
   - Priority inversion has now occurred because task M preempted a task (H) with higher priority.
   - The blocking time for task H now depends on a task (M) with lower priority that does not use the protected object.
   - If task M should use another protected object there would also be a potential risk that deadlock could occur.

CHALMERS

Priority inversion



CHALMERS

Ceiling priorities

Priority inversion can be reduced with the aid of a mechanism called <u>ceiling priorities</u>.

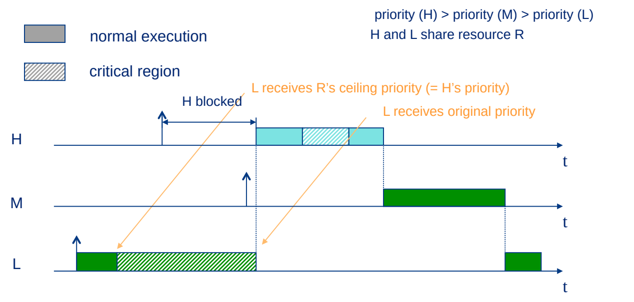
Each protected object is assigned a ceiling priority that is equal to the maximum priority among all tasks that may potentially request the protected object.

When a task executes the code of a protected object it is temporarily assigned a priority equal to that of the protected object's ceiling priority.

One method for ceiling priorities supported by Ada 95 is the Immediate Ceiling Priority Protocol (ICPP).

CHALMERS

Ceiling priorities in Ada 95 (ICPP)



CHALMERS

Ceiling priorities in Ada 95 (ICPP)

Besides minimizing priority inversion, ICPP exhibits some other nice properties in a single processor system:

- Mutual exclusion is guaranteed because a task that executes the code of a protected object cannot be preempted by any other task that also requests the protected object.
- A task can only be blocked once (in the beginning of its execution) by a task with lower priority.
- Freedom from deadlock is guaranteed if all objects are protected.

CHALMERS

Ceiling priorities in Ada 95

ICPP must be implemented in compilers that support *Annex D* (Real-Time Systems) in Ada 95.

A compiler vendor may choose to support multiple ceiling priority protocols.

Which ceiling priority protocol to use in Ada 95 is selected with the pragma `Locking_Policy`:

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

The identifier `Ceiling_Locking` corresponds to ICPP.

In Gnu Ada 95, the pragma is not needed as ICPP is the default policy.