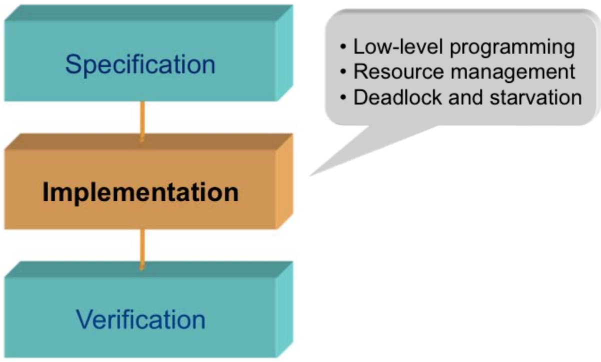
CHALMERS

Real-Time Systems



CHALMERS

Low-level programming

Low-level programming in Ada 95 enables writing *device drivers* for I/O circuits directly in a high-level language.

For systems programmed in a high-level language without support for low-level programming, device drivers must be written in the processor's assembly language.

Calling a device driver facilitates reading or writing data to/ from external units, e.g., hard disks, displays and keyboards.

A device driver conceals the details in the cooperation between software and hardware.

CHALMERS

Low-level programming

The programming language should make it possible to:

- Declare data types that enables manipulation of individual bits and bit strings.
- Define how declared variables are represented in the hardware.
- Read and write from/to hardware addresses where data and control registers of I/O circuits are located.
- Implement interrupt controlled I/O (i.e., associate hardware interrupts with high-level procedures for servicing the interrupt).

CHALMERS

Interrupt controlled I/O

Interrupt controlled I/O has the following advantages:

- Program controlled I/O uses ”polling”, which means that the processor spends most of its time in a ”busy-wait” loop.
- In many systems, one cannot afford to let the processor waste capacity in busy-wait loops. Interrupt controlled I/O avoids this.
- By activating the I/O handling code only when it is actually needed, it is easy to model a system event as a task.
- Depending on the activation pattern of the system event, it can be modeled as a periodic (e.g., interrupt from real-time clock) or aperiodic (e.g., network communication) task.

CHALMERS

Interrupt handling in Ada 95

Important guidelines for interrupt handling in Ada 95:

- Interrupts must be handled using protected objects.
- The interrupt service routine must be written as a procedure in the protected object.
- Data being handled by the interrupt service routine must be stored in local variables in the protected object.
- Reading and writing such data from the program code must be done via calls to functions, entries or procedures in the protected object.

CHALMERS

Interrupt handling in Ada 95

Procedure for implementing the interrupt handler:

1. Declare a protected object and write the interrupt service routine as a procedure in the protected object.
2. Inform the compiler that the procedure is an interrupt service routine, by adding the statement

```
pragma Interrupt_Handler(procedure_name);
```

   in the specification of the protected object.

3. Declare a variable and assign to it the logical number of the hardware interrupt signal. For example:

```
Int_ID : constant := Ada.Interrupts.Names.int_name;
```

CHALMERS

Interrupt handling in Ada 95

Procedure for implementing the interrupt handler (cont'd):

4. Associate the interrupt service routing with the logical number of the hardware interrupt signal, by calling the procedure

```
Attach_Handler(procedure_name'access, Int_ID);
```

5. Inform the compiler about the ceiling priority of the protected object, by adding the statement

```
pragma Interrupt_Priority(priority);
```

in the specification of the protected object.

The ceiling priority must be identical to the priority of the corresponding hardware interrupt signal.

CHALMERS

Interrupt handling in Ada 95

Why is it important that a ceiling priority is defined for the protected object?

- When an interrupt is requested, the processor hardware causes the interrupt service routine to be executed at a priority level associated with the interrupt signal.
- Functions, entries, and procedures in the protected object must execute at the same priority level as the interrupt service routine in order to preserve the mutual exclusion properties of the protected object.
- A task that calls a function, entry or procedure in the protected object temporarily assumes the ceiling priority while executing code in the protected object.

CHALMERS

Gnu Ada 95 M68K

Package **System** contains the following declarations:

```
subtype Any_Priority is Integer range 1..105;
subtype Priority is Any_Priority
    range Any_Priority'First .. 100;
subtype Interrupt_Priority is Any_Priority
    range Priority'Last .. Any_Priority'Last;
```

The priority of a protected object can be defined with

```
pragma Interrupt_Priority[(expression)];
```

Priority levels that are so high that they will mask (block) one or more hardware interrupt signals are of type **Interrupt_Priority**.

In Gnu Ada 95 M68K, the priority levels 101..105 correspond to the processor's (Motorola 68340) hardware priorities 1..5.

CHALMERS

Gnu Ada 95 M68K

Package **Ada.Interrupts** contains the following declarations:

```
package Ada.Interrupts is
  type Interrupt_ID is 64..80;
  ...
end Ada.Interrupts;
```

Package **Ada.Interrupts.Names** contains the following declarations:

```
package Ada.Interrupts.Names is
  TIMEINT    : constant Interrupt_ID := 64;
  ITIMERINT  : constant Interrupt_ID := 65;
  PORTBINT   : constant Interrupt_ID := 66;
end Ada.Interrupts.Names;
```

CHALMERS

Resource management

Resource management is a general problem that exists at several levels in a real-time system.

- The run-time system manages internal resources in the computer, e.g., CPU time, memory space, disks and communication channels.
- The application program manages other resources, that represents the controlled system, e.g., track sections in a train control system or robots in a manufacturing system:
  - Data structures and files
  - Sensors and actuators
  - Monitors and keyboards.

CHALMERS

Resource management

Classification of resources:

- Shared resources can be accessed by multiple users at the same time.
- Exclusive (non-shared) resources can only be accessed by one user at a time.
  - can be guaranteed with mutual exclusion
  - program code that is executed while mutual exclusion applies is called a critical region

CHALMERS

Resource management

Operations for resource management:

- **acquire:** to request access to a resource
- **release:** to release a previously acquired resource

The **acquire** operation can be either blocking or non-blocking:

- **Blocking:** the task that calls **acquire** is blocked until the requested resource becomes available.
- **Non-blocking:** **acquire** returns a status code that indicates whether access to the resource was granted or not.

The **acquire** operation can be generalized so that the calling task can provide a priority. The task with the highest priority will then be granted access to the resource in case of simultaneous requests.

CHALMERS

Example: resource handler

Problem: Write a protected object `One_Resource` that handles an exclusive resource.

- The protected object should have two entries, `Acquire` and `Release`.
- Via entry `Acquire` a task should be able to request access to the resource. If the resource is already being used, the task calling `Acquire` should be blocked.
- Via entry `Release` a task should be able to notify that it no longer needs the resource.

We solve this on the blackboard!

CHALMERS

Resource management

Problems with resource management:

- **Deadlock:** tasks blocks each other and none of them can use the resource.
  - Deadlock can only occur if the tasks require access to more than one resource at the same time
  - Deadlock can be avoided by following certain guidelines
- **Starvation:** Some task is blocked because resources are always assigned to other (higher priority) tasks.
  - Starvation can occur in most resource management scenarios
  - Starvation can be avoided by granting access to resources in FIFO order

In general, deadlock and starvation are problems that must be solved by the program designer!

CHALMERS

Deadlock

Example: Assume that two tasks, A and B, use two resources, R1 and R2. Each resource is handled by protected object `One_Resource`.

```
R1, R2 : One_Resource;

task A;
task body A is
begin
  R1.Acquire; -- task switch from A to B after this line causes deadlock
  R2.Acquire;
  ...         -- statements using the resources
  R2.Release;
  R1.Release;
end A;

task B;
task body B is
begin
  R2.Acquire;
  R1.Acquire;
  ...         -- statements using the resources
  R1.Release;
  R2.Release;
end B;
```

CHALMERS

Deadlock

Conditions for deadlock to occur:

1. Mutual exclusion
   – only one task at a time can use a resource
2. Hold and wait
   – there must be tasks that hold one resource at the same time as they request access to another resource
3. No preemption
   – a resource can only be released by the task holding it
4. Circular wait
   – there must exist a cyclic chain of tasks such that each task holds a resource that is requested by another task in the chain

CHALMERS

Deadlock

Guidelines for avoiding deadlock:

- Tasks should be either pure clients or pure servers
- Pure client tasks make calls to entries but do not have any entries themselves
- Pure server tasks have entries but do not make any calls to entries themselves
- Calls to entries during a rendezvous should be avoided

CHALMERS

Deadlock

Guidelines for avoiding deadlock (cont'd):

1. Task should, if possible, only use one resource at a time.
2. If (1) is not possible, all tasks should request resources in the same order.
3. If (1) and (2) are not possible, special precautions should be taken to avoid deadlock. For example, resources could be requested using non-blocking calls.

CHALMERS

Example: dining philosophers

The dining philosophers problem:

- Five Chinese philosophers sit at a round table.
- The philosophers alternate between eating and thinking. To be able to eat, a philosopher needs two sticks.
- There are only five sticks available: one stick between every pair of philosophers.
- Sticks are a scarce resource: only two philosophers can eat at the same time.
- How is deadlock and starvation avoided?

CHALMERS

Example: dining philosophers

The following solution will cause deadlock if all philosophers should take the left stick at exactly the same time:

```
loop
  Think;
  Take_left_stick;
  Take_right_stick;
  Eat;
  Drop_left_stick;
  Drop_right_stick;
end T1;
```

One way to avoid deadlock and starvation is to only allow four philosophers at the table at the same time.

CHALMERS

Example: dining philosophers

```
with Text_IO; use Text_IO;
procedure Philosopher_Demo is
  package Int_IO is new Integer_IO(Integer);
  use Int_IO;
  Max : constant Integer := 5;            -- five philosophers
  subtype Phil_No is Integer range 1..Max;

  protected type Room_t is                -- room type
    entry Enter;
    procedure Leave;
  private
    Places : Integer := Max - 1;          -- no more than four philosophers
  end Room_t;                             -- at the table simultaneously

  Room : Room_t;                          -- the room

  .
  .
  .
```

CHALMERS

Example: dining philosophers

```
.
.
.
protected type Stick_t is                    -- stick type
  procedure Set_ID(ID : in Phil_no);
  entry Take;
  entry Drop;
private
  MyID : Phil_no;
  Taken : Boolean := false;
end Stick_t;

Stick : array(Phil_No) of Stick_t;           -- the five sticks


task type Philosopher_t is                   -- philosopher type
  entry Start(ID : in Phil_no);
end Philosopher_t;

Philosopher : array(Phil_No) of Philosopher_t;  -- the five philosophers
.
.
.
```

CHALMERS

Example: dining philosophers

```
.
.
.
protected body Stick_t is

  procedure Set_ID(ID : in Phil_no) is
  begin
    MyID := ID;
  end Set_ID;

  entry Take when not Taken is
  begin
    Taken := true;
    Put("Stick"); Put(MyID, Width => 1); Put_Line(" taken");
  end Take;

  entry Drop when Taken is
  begin
    Taken := false;
    Put("Stick"); Put(MyID, Width => 1); Put_Line(" dropped");
  end Drop;

end Stick_t;
.
.
.
```

CHALMERS

Example: dining philosophers

```
.
.
.
protected body Room_t is

  entry Enter when Places > 0 is
  begin
    Places := Places - 1;
    Put_Line("One philosopher came");
  end Enter;

  procedure Leave is
  begin
    Places := Places + 1;
    Put_Line("One philosopher left");
  end Leave;

end Room_t;

.
.
.
```

CHALMERS

Example: dining philosophers

```
task body Philosopher_t is
  MyID : Phil_No;

  procedure Think is
  begin
    Put("Philosopher"); Put(MyID, Width => 1); Put_Line(" thinks");
    delay 3.0;
  end Think;
  procedure Eat is
  begin
    Put("Philosopher"); Put(MyID, Width => 1); Put_Line(" eats");
    delay 2.0;
  end Eat;
begin
  accept Start(ID : in Phil_No) do
    MyID := ID;
  end Start;
  loop
    Think;
    Room.Enter
    Sticks(MyID).Take; Sticks((MyID mod Max)+1).Take;
    Eat;
    Sticks(MyID).Drop; Sticks((MyID mod Max)+1).Drop;
    Room.Leave;
  end loop;
end Philosopher_t;
```

CHALMERS

Example: dining philosophers

```
begin

  for i in Phil_No loop
    Stick(i).Set_ID(i);
  end loop;

  for i in Phil_No loop
    Philosopher(i).Start(i);
  end loop;

end Philosopher_Demo;
```