Interrupts in Ada95 and the lab assignment

- Interrupts in Ada95
- Assignment 30
- Lab assignments Issues

Interrupts in Ada95

- An ***Interrupt*** represents a class of events that are detected by the hardware or system software.
- The *Occurrence* of an interrupt consists of its ***Generation*** and its ***Delivery***.
- The *Generation* of an interrupt is the event in the underlying hardware or system which makes the interrupt available to the program.
- *Delivery* is the action which invokes a part of the program (called the interrupt ***handler***) in response to the interrupt occurrence.
- In between the generation of the interrupt and its delivery, the interrupt is said to be *pending*. The handler is invoked once for each delivery of the interrupt.
- While an interrupt is being handled, further interrupts from the same source are *blocked*.

Interrupts in Ada95

- Certain interrupts are *Reserved*. The programmer is not allowed to provide a handler for a reserved interrupt.
- Each non-reserved interrupt has a default handler that is assigned by the run-time system.
- We as programmers will use non-reserved interrupts (specifically interrupt for PortB of MC68).

Imporatant Points About Interrupt

- In Ada interrupts are handled using ***protected object***.
  - Interrupt handlers are procedures (of course blocking) of the protected objects
- Data handled by *interrupt routine must be stored in local variables* of protected object.
- Reading/writing such data is done using calls to functions/entry/procedure of the protected object.
  - For example, reading/writing the data and status register must be through protected object. [think about *swrite()/waitsensor()* procedure in command.ads]
- The Three Numbers:
  - Interrupt Priority/Level (Specific to a hardware, already initialized)
  - Protected Object (used for interrupt handling) priority (to be initialized by us)
  - Interrupt Number (interrupt vector) (already initialized, but we will use it)
- Ada Hardware Interrupt Priority And Protected Object priority should be related
  - GNU Ada95 M68K, the priority level 101..105 maps to hardware priority 1..5.
- At port B the hardware interrupt has priority 4. *What is the protected object priority?*
- What is the interrupt number? For port B is it is 66 (defined in Ada package).

Steps for interrupt handler Installation(from Lecture 6)

- Step 1: Declare a protected object with a handler procedure (*procedure_name*).
  - Partial Definition is given in "command.adb" file.

    Now, **What you do to say that we have an interrupt handler?**
- Step 2: Inform compiler about the service by
  - pragma Interrupt_Handler(*procedure_name*) in the **specification** part.

    In "command.adb" it is given as: ***procedure handler;***

    Now, **What about interrupt vector?**
- Step 3: Declare a variable to store the logical number of hardware interrupt signal.
  - Int_ID: constant:=Ada.Interrupt.Names.PORTBINT; (Already defined in "traintypes.ads" in variable "***ivector***"). **What you do for this?**
- Step 4: Associate the handler and interrupt signal (Installation).
  - Attach_Handler (procedure_Name'access, Int_ID);

    What about the ceiling priority?
- Inform compiler the ceiling priority of the protected object in the **specification** by.
  - Pragma Interrupt_Priority(priority)?
  - **What is the value of ceiling priority?**

Interrupts in Ada95

Ada provides two styles of interrupt-handler installation and removal: *static* and *dynamic*.

In the static style, an interrupt handler in a given protected object is implicitly installed when the protected object comes into existence (is created), and the treatment that had been in effect beforehand (possibly the default handler) is implicitly restored when the protected object ceases to exist (is destroyed).

In the dynamic style, interrupt handlers are installed explicitly by procedure calls, and handlers that are replaced are not restored except by explicit request .

Interrupts in Ada95, example: static style

```
protected Static_style_Interrupt is
    ...
    procedure Our_Interrupt_Handler;
    Int_Id: Constant := implementation defined;
    pragma Attach_Handler( Our_Interrupt_Handler, Int_Id );
    ...
end Static_style_Interrupt;
```

- Connect the procedure Our_Interrupt_Handler to interrupt vector Int_Id.

Interrupts in Ada95, example: dynamic style

```
protected Dynamic_style_Interrupt is
    ...
    procedure Our_Interrupt_Handler;
    Int_Id: Constant := implementation defined;
    pragma Interrupt_Handler( Our_Interrupt_Handler );
    ...
end Dynamic_style_Interrupt;

protected body Dynamic_style_Interrupt is
    ...
begin
    Attach_Handler(Our_Interrupt_Handler , Int_Id );
end;
```

Interrupts in Ada95, protected object priority

- *Protected object priority* is the priority ceiling (Ada priority) used when any procedure, function or entry within the object is executed.
- This priority is normally not *the same* as the hardware interrupt priority, but they are *strongly related* and must match.

```
protected Any_style_Interrupt is
      ...
      Protected_object_priority : constant := implementation defined;
      pragma Interrupt_Priority( Protected_object_priority );
      ...
end Any_style_Interrupt;
```

Summary of implementation defined features, gada68k (used in the train lab.)

Binding interrupt :

```
Interrupt_PortB : constant := Ada.Interrupts.Names.PORTBINT;
-- is the vector interrupt number (0x42)...
-- => interrupt vector address 0x108.
      ...
```

Ada priority for protected object (given, Interrupt_PortB)

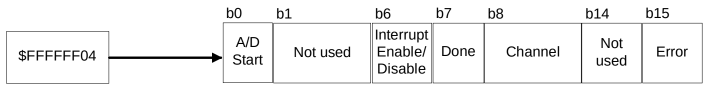
```
Protected_object_priority : constant := 104;
```

Corresponding hardware interrupt = Ada Priority - 100

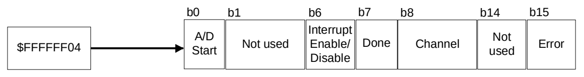
Assignment 30

Assume an eight bit register available at address FFFFFF04h in memory space (se below).

a) Define an appropriate type for this register layout.
b) Write a package `Cntrl_Reg` with functions `Read_Done`, returning the *Done*-bit, and `Read_Error` returning the *Error*-bit.
c) Add to the package a procedure `Write_Reg(FLAG:FLAG:CHAN_TYPE)` that updates the fields: *A/D Start*, *Interrupt Enable/Disable* and *Channel* simultaneously. Use value 0 for the read only bits *Done* and *Error*.



Sol. a)

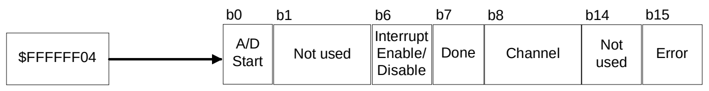


```
type FLAG is (CLEAR,SET);
for FLAG use (CLEAR => 0, SET => 1); -- Enumeration clause
for FLAG'Size use 1;           -- Size clause

type CHAN_TYPE is range 0..63;
for CHAN_TYPE'Size use 6;     -- "Bit field"  CHAN_TYPE needs 6 bits
type Control_Register is  -- A suitable control register definition
    record
          AD_Start:   FLAG;
          Int_Enable: FLAG;
          Done:       FLAG;
          Channel:    CHAN_TYPE;
          Error:      FLAG;
    end record;
-- A record representation clause is used to define actual bit positions (bit-field position)
for Control_Register use
    record
          AD_Start    at 0 range 0..0;
          Int_Enable  at 0 range 6..6;
          Done        at 0 range 7..7;
          Channel     at 0 range 8..13;
          Error       at 0 range 15..15;
    end record;
-- Tell compiler the size of this register with a size clause:
for Control_Register'Size use 16;       -- Type requires 16 bits
                                        -- undefined bits are not used
```

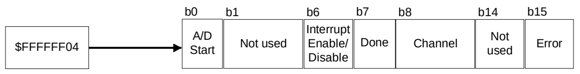
Assignment 30 (Cont.)

Assume an eight bit register available at address FFFFFF04h in memory space (se below).

a) Define an appropriate type for this register layout.
b) Write a package `Cntrl_Reg` with functions `Read_Done`, returning the *Done*-bit, and `Read_Error` returning the *Error*-bit.
c) Add to the package a procedure `Write_Reg(FLAG:FLAG:CHAN_TYPE)` that updates the fields: *A/D Start*, *Interrupt Enable/Disable* and *Channel* simultaneously. Use value 0 for the read only bits *Done* and *Error*.



Sol. b)



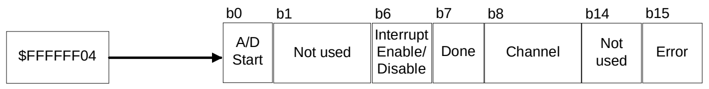
Write a package `Cntrl_Reg` with functions `Read_Done`, returning the *Done*-bit, and `Read_Error` returning the *Error*-bit.

```
with System, System.Storage_elements;
use System;
package body Cntrl_Reg is
   -- Declare the register at FFFFFF04h
   C_reg: Control_Register;
   for C_reg'address use: constant Address :=
                Storage_elements.to_address(16#FFFFFF04#);

   function Read_Done return FLAG is
   begin
        return(C_reg.Done);
   end Read_Done;

function Read_Error return FLAG is
   begin
        return(C_reg.Error);
   end Read_Error;
end Cntrl_Reg;
```

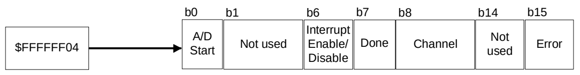
Assignment 30 (Cont.)

Assume an eight bit register available at address FFFFFF04h in memory space (se below).

a) Define an appropriate type for this register layout.
b) Write a package `Cntrl_Reg` with functions `Read_Done`, returning the *Done*-bit, and `Read_Error` returning the *Error*-bit.
c) Add to the package a procedure `Write_Reg(FLAG:FLAG:CHAN_TYPE)` that updates the fields: *A/D Start*, *Interrupt Enable/Disable* and *Channel* simultaneously. Use value 0 for the read only bits *Done* and *Error*.



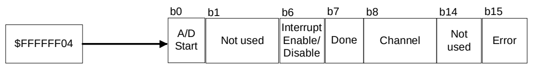
Sol. c)



Add to the package a procedure `Write_Reg(FLAG:FLAG:CHAN_TYPE)` that updates the fields: *A/D Start*, *Interrupt Enable/Disable* and *Channel* simultaneously. Use value 0 for the read only bits *Done* and *Error*.

```
procedure Write_Reg( ADS_Flag, I_ED_Flag: in FLAG;
               Ch: in CHAN_TYPE) is

   Shadow_Register: Control_Register;

begin
   Shadow_Register:= (AD_Start => ADS_Flag,
                      Int_Enable => I_ED_Flag,
                      Done => CLEAR,
                      Channel => Ch,
                      Error => CLEAR);
   C_reg := Shadow_Register;  -- Register update
end Write_Reg;
```

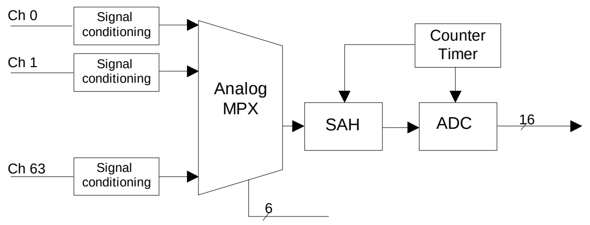
Assignment 30



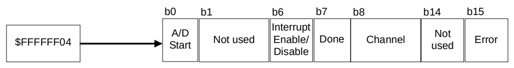
d)

The register is a control register for an A/D converter where bits have the following functions:

A/D Start	Set to 1 to initiate conversion.
Interrupt Enable/Disable	Set to 1 if the finalized conversion should generate an interrupt
Done	If this bit is zero, the conversion is still pending, otherwise it's ready
Channel	Chooses one out of the 64 analog inputs.
Error	This bit is clear if the conversion was ok, otherwise it is set.



Assignment 30



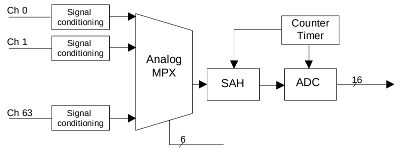
When a conversion is initiated the converter samples the value from Channel.

The Done bit is reset. The sampled value is converted to a binary value and placed in a 16 bit data register located at FFFFFF02. The Done bit is now set. If the Interrupt Enable/Disable is set an interrupt is generated.

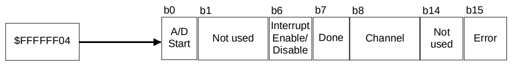
Write an ADA package AD_Converter with a single procedure, using Interrupt Enable:

```
procedure Read_AD(Ch: in CHAN_TYPE; M:out MEASUREMENT ;
      AD_busy: out BOOLEAN);
```

If the conversion was successful, then `M` holds the converted value and *AD_Busy* is FALSE. If the converter is busy, then *AD_Busy* is TRUE and `M` is undefined. If a conversion error occurs, then exception `Conversion_Error` should be raised.



Sol. d)



```
-- Package specification *.ads
package AD_Converter is
   Max_Measure : constant := (2**16)-1;  -- 16 bits reg
   subtype MEASUREMENT is Integer range 0..Max_Measure;
   type Chan_Type is range 0..63; -- 64 channels
   procedure Read_AD(Ch: in CHAN_TYPE; M:
        out MEASUREMENT ; AD_busy:BOOLEAN);
   Conversion_error : exception;
end AD_Converter;
```

Sol. d)

```
-- Package body *.adb (the structure...)
with System, System.Storage_elements, Ada.Interrupts,
        Ada.Interrupts.Names;
use System, System.Storage_elements, Ada.Interrupts,
        Ada.Interrupts.Names;
package body AD_Converter is
    -- type declarations goes here, see (a ...
    -- register and priority declarations  goes here
   protected type AD_Device_Interrupt is
        entry wait_for_completion(M: out MEASUREMENT);
        pragma Interrupt_Priority(AD_Dev_priority); -- object priority
        procedure Handler;
        pragma interrupt_handler(Handler);

        Interrupt_Occured : Boolean := False;
   end AD_Device_Interrupt;

   protected body AD_Device_Interrupt is
        -- protected body goes here
   end AD_Device_Interrupt;
   AD_Interrupt : AD_Device_Interrupt;
■       Int_Id: Constant := --implementation defined HW interrupt signal;

begin
   Attach_Handler( AD_Interrupt.Handler'Access, Int_Id);
end AD_Converter;
```

Sol. d)

```
    -- register and priority declarations:
   C_reg: Control_Register;
   for C_reg'address use constant Address :=
              to_address(16#FFFFFF04#);
   D_reg: MEASUREMENT;
   for D_reg'address use constant Address :=
              to_address(16#FFFFFF02#);

-- Priorities are implementation defined
-- In this implementation object priority 104 corresponds to
-- hardware priority 4

   AD_Dev_priority: constant := 104;
```

Sol. d)

```
protected body AD_Device_Interrupt is
     procedure Handler is
     begin   -- Interrupt
             Interrupt_Occured := True;
     end Handler;

     entry wait_for_completion (M: out MEASUREMENT)
           when Interrupt_Occured is     -- Wait_for_Interrupt
     begin
           if C_reg.Done = SET and C_reg.Error = CLEAR then
                 -- C_Reg_OK
                 M := D_reg;
           else
                 -- C_Reg_Not_OK
                 interrupt_occured := FALSE;
                 raise Conversion_error;
           end if;
           interrupt_occured := FALSE;
     end wait_for_completion;
end AD_Device_Interrupt;
```

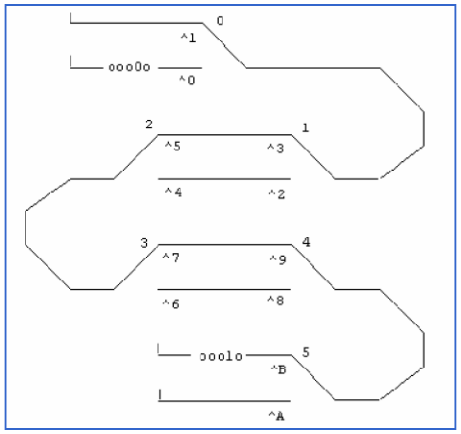
Sol. d)

```
procedure Read_AD(Ch: in CHAN_TYPE; M:
   out MEASUREMENT ; AD_busy:BOOLEAN) is

begin
   if C_reg.Done = CLEAR
        AD_Busy := TRUE;
   else
        Write_Reg(1, 1; Ch); -- see c)
        wait_for_completion( M );
        AD_Busy := FALSE;
   end if;

end Read_AD;
```

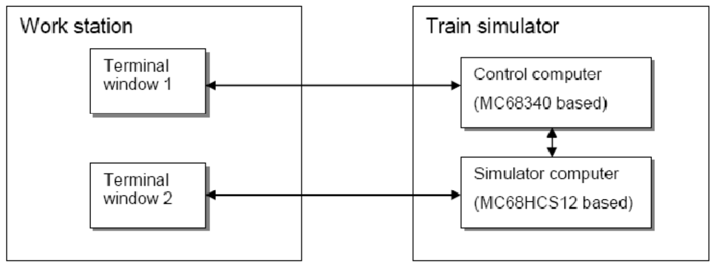
```
package body AD_Converter is

C_reg: Control_Register;
for C_reg'address use: constant Address :=
          to_address(16#FFFFFF04#);

D_reg: MEASUREMENT;
for D_reg'address use constant Address :=
                  to_address(16#FFFFFF02#);

AD_Dev_priority: constant := implementation
defined

protected type AD_Device_Interrupt is

entry wait_for_completion(M: out MEASUREMENT);
procedure Handler;
pragma Interrupt_Priority(AD_Dev_priority);

pragma Interrupt_handler(Handler);
Interrupt_Occured : Boolean := False;
end AD_Device_Interrupt;

protected body AD_Device_Interrupt is
      procedure Handler is
        begin        -- Interrupt
            Interrupt_Occured := True;
        end Handler;
entry wait_for_completion (M: out MEASUREMENT)  when Interrupt_Occured
is
  begin
      if C_reg.Done = SET and C_reg.Error = CLEAR then
           -- C_Reg_OK
           M := D_reg;
      else
           -- C_Reg_Not_OK
           interrupt_occured := FALSE;
           raise Conversion_error;
       end if;
       interrupt_occured := FALSE;
end wait_for_completion;

end AD_Device_Interrupt;

procedure Read_AD(Ch: in CHAN_TYPE; M:
           out MEASUREMENT ; AD_busy:BOOLEAN) is
begin
        if C_reg.Done = CLEAR
           AD_Busy := TRUE;
        else
           Write_Reg(1, 1; Ch); -- see c)
           wait_for_completion( M );
           AD_Busy := FALSE;
        end if;
end Read_AD;
  AD_Interrupt : AD_Device_Interrupt;
  Int_Id: Constant := --implementation defined ;

begin
           Attach_Handler(AD_Interrupt.Handler'Access,Int_Id);
end AD_Converter;
```

Lab assignment

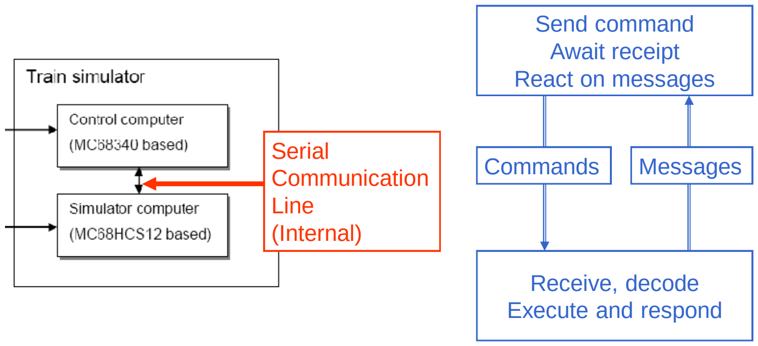
- Train simulator overview
- Simulator commands
- Simulator messages



Overview

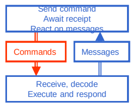

Work station
Terminal window 1
Terminal window 2
Train simulator
Control computer (MC68340 based)
Simulator computer (MC68HCS12 based)

Train simulator



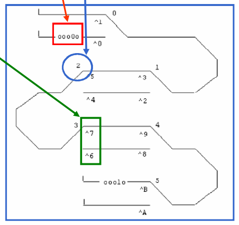
Command types

Configure	This command restarts the train simulator. Trains, exchanges and sensors will "momentarily" be set to their initial positions (one of the few occasions there the model surpasses reality in performance). The command makes it possible to choose between a few different configurations.
Set exchange	Set an exchange in the specified position.
Activate sensor	The specified sensor is activated or deactivated.
Set speed	The speed for the specified train is set to specified value.



Command codes

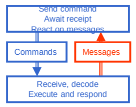
MSB	Bit number						LSB	Command type
7	6	5	4	3	2	1	0	
Value field						Type field		
0	Log	Rec.	C-number			0	0	Configure
0	Off	E-number				0	1	Set exchange
0	Active	S-number				1	0	Activate sensor
Rev	Speed		T-number			1	1	Set speed



Message types

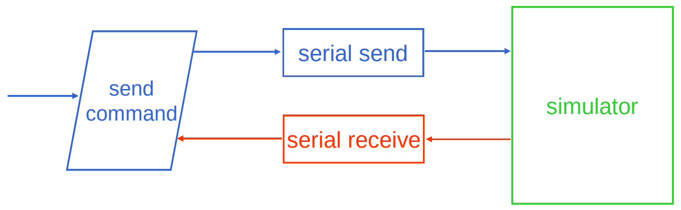
Receipt	The latest command of the specified type has been completed or alternatively rejected for some reason
Passage	A sensor reacts when a train arrives to or leaves the place where the sensor is located.
Catastrophe	A train has collided, deranged or reversed. When a catastrophe is reported the simulator has to be reconfigured.

Collision	A train may not be driven against another train or into a stop block.
Deranging	A train may not be driven into an exchange set in the wrong position nor can an exchange be repositioned while a train is passing.
Reversing	The direction of movement cannot be changed while a train is moving. To prevent a catastrophe the train first have to be stopped.
Addressing	A command has been given that references a component (train, exchange, sensor) that do not exist in the current configuration

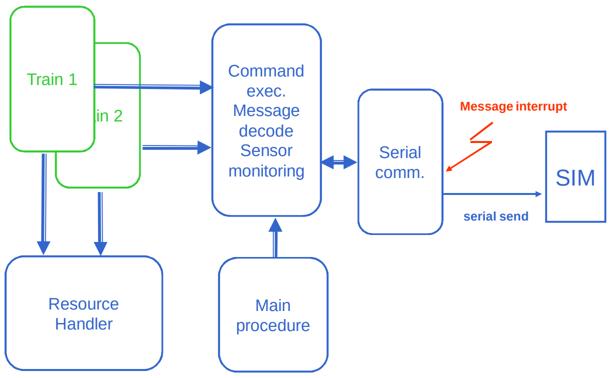


Controlling the simulator

- Every command requires a single byte transmission to the simulator.
- Every command must be acknowledged by the simulator (Receipt). Otherwise synchronisation is impossible.



Train Application



Summary

- Details about Interrupts in Ada95
- Demonstration of Assignment 30
- Recommended home work: Assignment 29 and preparations for the train lab (Serial device driver)