Low level programming and Exception handling in *Ada95*

- Assignment 8 (from last exercise class)
- Big and Little Endian representation
- Assignments 27 and 28
- Exception handling, short presentation
- Assignments 23
- Lab Related Issues

Assignment 8

```
Three integer variables is shared by several tasks. Write an ADA package
Notice_Board containg read and write operations by concurrent tasks, for
these variables. The following type is declared:

type var_num is range 1 .. 3;

The package shall include the following procedures:

procedure read (num : in var_num; value : out integer)
        -- Returns value of variable denoted by 'var_num'.
        -- Block the calling task if the variable

        -- not have been previously assigned through 'write'.

procedure write (num : in var_num; value : in integer)
        -- Assign 'value' to variable denoted by 'var_num'.

Write operations are mutual exclusive for a particular variable, i.e writing
one variable should not block operations on another variable. Hint: Create a
protected object for each variable).
```


Step 1, the specification

```
package Notice_Board is
   type Var_Num is range 1 .. 3;
   procedure read( Num : in Var_Num; Value : out Integer);
   procedure write( Num : in Var_Num; Value : in Integer);
end Notice_Board;
```

Goes to specification file, e.g. "`Notice_Board.ads`"

Declarations are visible throughout the application

Step 2, declarations

Goes to declaration file, e.g. "`Notice_Board.adb`"
Details the implementation, also contains locals i.e. visible inside but not outside the package and "privates", i.e. unique copies for every object instance.

```
package body Notice_Board is
   protected type Protected_Int is
      entry Read( Value : out Integer);
      procedure Write( Value : in Integer);
   private
      X : Integer := 0;
      Written : Boolean := False;
   end Protected_Int;

   protected body Protected_Int is
   -- implementation of entry 'Read' and local procedure 'Write' (protected)
   end Protected_Int;
---------------------------------------------------------------------------
   -- multiple instances of the protected object...
   type Protected_Int_List is array (Var_Num) of Protected_Int;
   Board_Variables : Protected_Int_List;
 -- Exported (visible) procedures
   procedure read( Num : in Var_Num; Value : out Integer) is
   ... -- implementation of procedure 'Read' globally visible
   procedure write ( Num : in Var_Num; Value : in Integer) is
   ... -- implementation of procedure 'Write' globally visible
end Notice_Board;
```

Step 3, protected details

```
protected body Protected_Int is

    entry Read(Value : out Integer) when Written is
    begin
       Value := X;
    end;

    procedure Write(Value : in Integer) is
    begin
       X := Value;
       Written :=  True;
    end;

 end Protected_Int;
```

Note that '`Value`' is unique for every instance of the `Protected_Int` object.

The choice of an *entry* for `Read` is motivated by the required guard (`Written`).

Step 4, globally visible procedure details

```
procedure Read( Num : in Var_Num; Value : out Integer) is
 begin
     Board_Variables(Num).Read(Value);
 end;

 procedure Write ( Num : in Var_Num; Value : in Integer) is
 begin
     Board_Variables(Num).Write(Value);
 end;
```

'`Num`' indicates the actual instance of the protected object


```
package Notice_Board is
   type Var_Num is range 1 .. 3;
   procedure read( Num : in Var_Num; Value : out Integer);
   procedure write( Num : in Var_Num; Value : in Integer);
end Notice_Board;
```

```
package body Notice_Board is

  protected type Protected_Int is
     entry Read( Value : out Integer);
     procedure Write( Value : in Integer);
  private
     X : Integer := 0;
     Written : Boolean := False;
  end Protected_Int;
```

```
  protected body Protected_Int is
     entry Read(Value : out Integer) when Written is
       begin
         Value := X;
       end;
      procedure Write(Value : in Integer) is
       begin
         X := Value;
         Written :=  True;
       end;
  end Protected_Int;
```

```
type Protected_Int_List is array (Var_Num) of Protected_Int;
Board_Variables : Protected_Int_List;
```

```
procedure read( Num : in Var_Num; Value : out Integer) is
   begin
      Board_Variables(Num).Read(Value);
  end;
procedure write ( Num : in Var_Num; Value : in Integer) is
begin
      Board_Variables(Num).Write(Value);
end;
end Notice_Board;
```


Recommended home work...

Eloborate on the following assignments..

1-5

Will get you started and going with the IDE and ada95 taking mechanisms.

6

Learn how to make a set of procedures "generic" simply by using *types*.

7

A simple exercise on protected objects.

10

Preparations for the laboratory assignments.

Big and Little Endian representation

- ***Endianness*** is the ordering used to represent some kind of data.
- Let a 8-bit register as follows:

bit	bit	bit	bit	bit	bit	bit	bit
7	6	5	4	3	2	1	0

- **Inside Memory how to represent the nibbles (4 bits): 2 ways**

Little Endian

bit	bit	bit	bit	bit	bit	bit	bit
7	6	5	4	3	2	1	0

Big Endian (ADA 95)

bit	bit	bit	bit	bit	bit	bit	bit
7	6	5	4	3	2	1	0

Assignment 27

```
Assume the following declarations:
-- nybble.ads
with System.Storage_elements;
package NYBBLE is
    type BYTE is range 0..255; -- Named type and min..max values
    type HIGH_NIBBLE_TYPE  is range 0..15; -- Named type and min..max values
    type LOW_NIBBLE_TYPE is range 0..15; -- Named type and min..max values
    type NIBBLES  is
    record
      high_nibble: HIGH_NIBBLE_TYPE;
      low_nibble : LOW_NIBBLE_TYPE;
    end record;
    for NIBBLES use
    record
      high_nibble at 0 range 0..3;        -- means b7-b4 in big endian
      low_nibble  at 0 range 4..7;        -- means b3-b0 in big endian
    end record;

    D_reg: BYTE;
    for D_reg'address use constant System.address :=
      System.Storage_elements.to_address(16#FFFFFF15#);
    procedure wnibble ( W : HIGH_NIBBLE_TYPE );
    procedure wnibble ( W : LOW_NIBBLE_TYPE );
end NYBBLE;
```


Assignment 27, cont'd

We now want a "single" procedure `wnibble(..)` to write either the high nybble or the low nybble of a byte to the register located at FFFFFF15. Show how to do this using function overloading and unchecked conversions.

Solution 27 (nybble.adb)

```
with unchecked_conversion;
package body NYBBLE is

    function to_byte is new unchecked_conversion( LOW_NIBBLE_TYPE, BYTE );
    function to_byte is new unchecked_conversion( HIGH_NIBBLE_TYPE, BYTE );


    procedure wnibble ( W : LOW_NIBBLE_TYPE ) is
      begin
            D_reg := to_byte( W );
      end;


    procedure wnibble ( W : HIGH_NIBBLE_TYPE ) is
      begin
            D_reg := to_byte( W );
      end;

end NYBBLE;
```

Assignment 28

Assume two eight bit registers available at address FFFFFF03h and FFFFFFF05h in memory space.

The first register, called DATA, holds a character supplied by an external device.

The second register STATUS has a single read-only "sticky-bit" RxRdy
which is set (1) each time the data register is filled with a new value
the bit is reset (0) by the peripheral device when the data register is read.

Remaining bits in this registers are always read as 0.

Write a

```
procedure ReadRegister ( valid : out BOOLEAN; data :out BYTE)
```

that either returns with "fresh" data (valid=TRUE) or "old" data (valid=FALSE).

Solution 28

```
type BYTE is range 0..255;
DATA, STATUS : BYTE;

for DATA'address use constant System.address := System.Storage_elements.to_address(16#FFFFFF03#);
for STATUS'address use constant System.address :=System.Storage_elements.to_address(16#FFFFFF05#);

pragma Volatile( STATUS );
pragma Volatile( DATA );

procedure ReadRegister(valid : out BOOLEAN; data: out BYTE) is
begin
    if STATUS /= 0
            -- "fresh" data
            valid := TRUE;
    else
            valid := FALSE;
    end if;
    data = DATA;
end ReadRegister;
```

Pragma Volitile(variable_name) enables compiler to supress optimization

Exception handling

```
procedure X is
      begin
      -- your code goes here as usual
      exception
            when Some_Exception =>
                  Do_This;
      end X;
```

Your program should be designed to handle even the unlikely events.

Some_Exception

Exceptions are either system defined or application defined. Important system defined exceptions are:

- **Constraint_Error** - This will occur if something goes out of its assigned range.
- **Numeric_Error** - This will occur if something goes wrong with arithmetic such as the attempt to divide by zero.
- **Program_Error** - This will occur if we attempt to violate an Ada control structure such as dropping through the bottom of a function without a return.
- **Storage_Error** - This will occur if we run out of storage space through either recursive calls or storage allocation calls.
- **Tasking_Error** - This will occur when attempting to use some form of tasking in violation of the rules.

Some_Exception

While system defined exceptions are pre-declared, your application defined exception has (of course) to be declared:

```
procedure X is
      My_Own_Exception : exception;
      begin
      -- your code goes here as usual
      exception
            when My_Own_Exception =>
                  Do_This;
      end X;
```


Raising exceptions

System defined exceptions are normally raised by the run-time (or operating) system. For example:

```
procedure X is
      ...
      begin
            A := B/C;   -- what if C = 0 ?
      exception
            when Numeric_Error =>
                  Do_This;
      end X;
```

Execution is aborted when C = 0. Since there is an exception handler 'Do_This' will immediatly be executed.

Raising exceptions

While system defined exceptions are raised by the run-time (or operating) system, any exception can be raised by a program:

```
procedure X is
      ...
      My_Own_Exception : exception;
      begin
            if  C = 0
                  raise My_Own_Exception
            A := B/C;
      exception
            when My_Own_Exception =>
                  Do_This;
      end X;
```


Unhandled exceptions

Any exception, not handled within the scope it occured, will be propagated to the next higher level.

```
procedure main is
      ...
      Y;
end main;
```

```
procedure Y is
      ...
      X;
end Y;
```

```
procedure X is
      ...
      A:=B/C;
end X;
```

It will, if not handled by the application propagate to the system, resulting in some confusing printout such as `'Unhandled exception, program terminated'.`

Unhandled exceptions

As a minimum requirement, your top level procedure should handle any exception.

```
procedure main is
      ...
      Y;

      exception
            when ...
end main;
```

Resulting in a (hopefully) less confusing printout.

Simple exception handling

Exception handling is strictly application dependant. But at early software development stages, a simple printout is sufficient

```
exception
      when Error : E1 |E2 ... =>
            Put ("The exception was ");
            Put_Line ( Exception_Name(Error) );
```

`Ada.Exceptions` defines a data type called `Exception_Occurrence` and provides a function called `Exception_Name` which produces the name of the exception as a string from an `Exception_Occurrence`.

Exceptions during elaboration

Exception handling can inhibit the execution of a procedure or a function, consider the following example:

```
procedure Impossible is
      VALUE : constant := 8;
      subtype LIMIT_RANGE is INTEGER range 14..33;
      Funny : LIMIT_RANGE := VALUE;
   begin
      Put_Line("You will never see this printout");
   exception
      when Constraint_Error =>
              Put_Line("Constraint error occurred");
   end Impossible ;
```


Assignment 23.

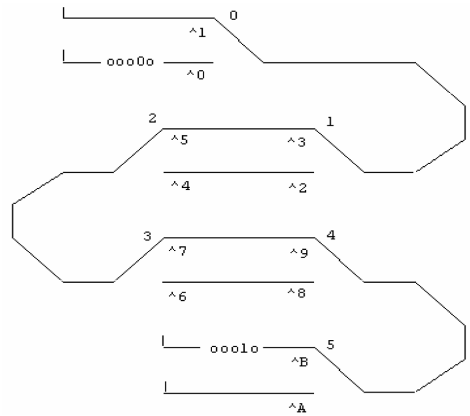
Ada95 allows the application programmer to define any handling of exceptional events. Give an example of how you, as the programmer should handle the first instance of a particular exception, but would propagate a second occurrence of the same exception.

Assignment 23.

Proposed solution:

```
...
  exception
     when My_Recoverable_Exception =>
        begin -- attempt recovery
            Recover;
            exception
                 when My_Recoverable_Exception =>
                 Abandon; -- recovery failed!
        end;
```


Lab Related Issues…



- How many tracks?
- What are the shared tracks?
- Resource Handler and Exceptions

Recommended home work...

```
21
Exception handling (provide diagnostics).
24,25,26
Type declarations and basic IO programming.
```