SPARK

Quick Reference 2

Patterns

Main Program

The following outlines a 'complete' framework for a generic control program (when combined with the input/output packages which follow) - it *could* actually be analysed by the Examiner to ensure that there are no unexpected dependencies in the information flow.

```
package Monitor
is
  type Limit is limited private;
  …
end Monitor;

with Sensor, Valve, Monitor;
--# inherit Sensor, Valve, Monitor;
--# main_program;
procedure Main
--# global in Sensor.State;
--#        out Valve.State;
--# derives Valve.State from Sensor.State;
is
  ControlLimit : Monitor.Limit;

  procedure Control
  --# global in Sensor.State;
  --#        out Valve.State;
  --#        in out ControlLimit;
  --# derives Valve.State,
  --#         ControlLimit from Sensor.State,
  --#                           ControlLimit;
  is
  …
  end Control;

begin
  …
  loop
    Control;
  end loop;

end Main;
```

External Variables

These are variables used for communication with the world outside the SPARK program. They are denoted in SPARK by giving a mode to the package own variable. Each may be an input **or** an output but **not** both. The Examiner recognises that:

- the values of inputs may change between reads;
- consecutive writes to outputs are not ineffective.

To prevent various problems arising with external variables:

- external in variables may not be updated;
- external out variables may not be read;
- external variables may only appear directly in return statements and assignment statements;
- external variables may not appear as part of a composite expression.

Inputs

The following package might be used to read from an input.

```
package Sensor
--# own in State;            -- external in variable
is
  function Read return Integer;
  --# global in State;
end Sensor;

package body Sensor
is
  State : Integer;
  for State'Address use 16#ffff_1234#;

  function Read return Integer
  is
  begin
    return State; -- You may need to use 'Valid
                  -- here check check data OK!
  end Read;

end Sensor;
```

Outputs

The following package might be used to write to an output.

```
package Valve
--# own out State;           -- external out variable
is
  procedure Write(Valve_Setting: in Integer);
  --# global out State;
  --# derives State from Valve_Setting;
end Valve;

package body Valve
is
  State : Integer;
  for State'Address use 16#ffff_1238#;

  procedure Write(Valve_Setting: in Integer)
  is
  begin
    State := Valve_Setting;
  end Write;

end Valve;
```

Child Packages

Private Child Packages

Private children offer a natural way of achieving encapsulation and top-down refinement of program state - this is a neat alternative to the use of embedded packages.

```
package Controls
--# own in State;
is
  type Buttons is (Pressed, NotPressed);

  procedure ReadMode(Setting : out Buttons);
  --# global in State;
  --# derives Setting from State;
  …
end Controls;

--# inherit Controls;
private package Controls.Mode
--# own in State;
is
  procedure Read(Setting : out Controls.Buttons);
  --# global in State;
  --# derives Setting from State;
end Controls.Mode;

with Controls.Mode;
package body Controls
--# own State is in Controls.Mode.State;
is
  procedure ReadMode(Setting : out Buttons)
  --# global in Mode.State;
  --# derives Setting from Mode.State;
  is
  begin
    Mode.Read(Setting);
  end ReadMode;
  …
end Controls;
```

Notes:

- a private child *is* visible to its parent's body,
- a private child is *not* visible to external library packages,
- private children *can* see the specification of their parent but *cannot* call parent subprograms or refer to their abstract own variables,
- all variables within the own variable clause of a private child must appear as constituents in a refinement clause in the body of the parent,
- the initialisation specification of a private child must be consistent with that of its parent.

Public Child Packages

Public children allow the facilities of a package to be extended without the need to alter the package itself - thereby avoiding the need for recompilation and re-testing.

```
package Parent
is
  type T is private;

  procedure OpP(X : in out T);
  --# derives X from X;

  private
    type T is range 0..1000;

end Parent;

--# inherit Parent;
package Parent.Child
is
  procedure OpC(X : in out Parent.T);
  --# derives X from X;

end Parent.Child;

package body Parent.Child
is
  procedure OpC(X : in out Parent.T)
  is
  begin
    Parent.OpP(X);
    X := X + 100;
  end OpC;
end Parent.Child;
```

Notes:

- a public child is *not* visible to its parent,
- a public child *is* visible to external library packages,
- public children *can* see the specification of their parent including any private parts,
- a public child is permitted to inherit packages not inherited by its parent,
- the *own variables* of a public child are completely independent from those of its parent.

Data Abstraction

Abstract Data Types

Abstract data types define objects with a set of operations that characterise the behaviour of those objects. They are constructed using packages with private types. A package implementing an ADT consists of two separate components - a specification (defining the object type and operations) and a body (containing implementation details hidden from package users).

```
package Stacks
is
  type Stack is limited private;

  function IsEmpty(S : Stack) return Boolean;
  function IsFull(S : Stack) return Boolean;

  procedure Clear(S : out Stack);
  --# derives S from ;

  procedure Push(S : in out Stack; X : in Integer);
  --# derives S from S, X;

  procedure Pop(S : in out Stack; X : out Integer);
  --# derives S, X from S;

  private
    StackSize : constant := 100;
    type PtrRange is range 0..StackSize;
    subtype IdxRange is PtrRange range 1..StackSize;
    type Vector is array(IdxRange) of Integer;

    type Stack is
    record
      StackVector  : Vector;
      StackPointer : PtrRange;
    end record;

end Stacks;

package body Stacks
is
  procedure Push(S : in out Stack; X : in Integer)
  is
  begin
    S.StackPointer := S.StackPointer + 1;
    S.StackVector(S.StackPointer) := X;
  end Push;
  ...
end Stacks;
```

Abstract State Machines

Whereas an ADT package gives the ability to declare objects and then operate on them, an abstract state machine package declares just one object and its operations. An ASM can be represented by a package with variables which record its state declared in its body. Procedures which act on the machine and functions that observe its state are specified in the visible part of the package specification.

```
package Stack
--# own State;                          -- abstract own variable
--# initializes State;
is
  procedure Push(X : in Integer);
  --# global in out State;
  --# derives State from State, X;

procedure Pop(X : out Integer);
  --# global in out State;
  --# derives State, X from State;

end Stack;

package body Stack
--# own State is S, Top;                        -- refinement
is
  StackSize : constant := 100;
  type TopRange is range 0..StackSize;
  subtype IndexRange is TopRange range 1..StackSize;
  type Vector is array(IndexRange) of Integer;
  S : Vector;
  Top : TopRange;

  procedure Push(X : in Integer)
  --# global in out S, Top;          -- refinement annotation
  --# derives S from S, Top, X &           -- also needed
  --#         Top from Top;              -- on subprograms
  is
  begin
    Top := Top + 1;
    S(Top) := X;
  end Push;

  procedure Pop(X : out Integer)
  --# global in out Top;
  --#        in S;
  --# derives Top from Top &
  --#         X from S, Top;
  is
  begin
    X := S(Top);
    Top := Top - 1;
  end Pop;

begin
  Top := 0;
  S := Vector'(others => 0);
end Stack;
```