15. Finite State Machines, Part 3

As you have seen in the previous chapters, interrupts provide a way to react to various events without the need to constantly monitor the environment to detect that the interesting event has taken place. In particular, we can use interrupts to react to buttons being pressed - this allows us to revisit the previous design for the fan control system, which was based on periodic button polling.

In its last version, the system relied on the following main program skeleton:

```
package body Program is

   -- definitions of types and constant array Transitions

   procedure Control_Motor (S : in Fan_State) is
   begin
      -- ...
   end Control_Motor;

   procedure Read_Buttons (B : out Buttons_State) is
   begin
      -- ...
   end Read_Buttons;

   procedure Run is
      Current_State : Fan_State;
      Buttons : Buttons_State;
   begin

      -- initialize the device
      -- ...

      -- repeat the control loop
      loop
         Read_Buttons (Buttons);

         Current_State := Transitions (Current_State, Buttons);

         Control_Motor (Current_State);

         Utils.Waste_Some_Time;
      end loop;
   end Run;

end Program;
```

In the above structure the loop is responsible for periodic polling of button states. We can refactor this loop by moving the button-handling part to a separate procedure called `Handle_Buttons`:

```
package body Program is

   -- ...

   Current_State : Fan_State;

   procedure Handle_Buttons is
      Buttons : Buttons_State;
   begin
      Read_Buttons (Buttons);

      Current_State := Transitions (Current_State, Buttons);

      Control_Motor (Current_State);
   end Handle_Buttons;

   procedure Run is
   begin

      -- initialize the device
      -- ...

      -- repeat the control loop
      loop
```

```
         Handle_Buttons;

         Utils.Waste_Some_Time;
      end loop;
   end Run;

end Program;
```

There is no need to call procedure `Handle_Buttons` repeatedly, we can rely on the peripheral subsystem to detect that any of the buttons was actually pressed, which can even ensure better responsiveness as the dead time between checks can be eliminated. This means that the `Handle_Buttons` procedure is a good candidate to become an interrupt handler. We will need to properly register this procedure as the handler in the linker script with some valid assigned external name and this assignment can take place in the specification of package `Program`, just as we have exercised in earlier examples.

Note also that `Current_State` was moved outside of the procedure `Handle_Buttons`, to the package level, to ensure that it retains its value between invocations of the handler - previously it was not needed, because button handling was entirely performed from within the same execution context. We will also change `Control_Motor` to a parameterless procedure - with shared `Current_State` there is no need to pass it around as a parameter any longer.

After all these changes, the complete content of file `program.ads` is:

```
package Program is

   procedure Run;
   pragma Export (C, Run, "run");

   procedure Handle_Buttons;
   pragma Export (C, Handle_Buttons, "handle_buttons");

end Program;
```

and the body of this package (file `program.adb`) is:

```
with Pins;
with Utils;

package body Program is

   type Fan_State is (Stop, Slow, Medium, Fast);

   Current_State : Fan_State;

   type Buttons_State is (None, Up, Down, Both);

   type Transition_Table is array (Fan_State, Buttons_State)
      of Fan_State;

   Transitions : constant Transition_Table :=
      (Stop   =>
         (None => Stop,   Up => Slow,   Down => Stop,   Both => Stop),
       Slow   =>
         (None => Slow,   Up => Medium, Down => Stop,   Both => Stop),
       Medium =>
         (None => Medium, Up => Fast,   Down => Slow,   Both => Stop),
       Fast   =>
         (None => Fast,   Up => Fast,   Down => Medium, Both => Stop));

   subtype Output_Pins is Pins.Pin_ID range Pins.Pin_4 .. Pins.Pin_6;

   type Output_Mapping_Array is array (Fan_State, Output_Pins)
      of Boolean;

   Outputs : constant Output_Mapping_Array :=
      (Stop   => (False, False, False),
       Slow   => (True,  False, False),
       Medium => (False, True,  False),
       Fast   => (False, False, True));

   procedure Control_Motor is
   begin
      Pins.Write (Pins.Pin_4, Outputs (Current_State, Pins.Pin_4));
      Pins.Write (Pins.Pin_5, Outputs (Current_State, Pins.Pin_5));
      Pins.Write (Pins.Pin_6, Outputs (Current_State, Pins.Pin_6));
   end Control_Motor;

   procedure Read_Buttons (B : out Buttons_State) is
      B_Down : Boolean;
      B_Up : Boolean;
   begin
      Pins.Read (Pins.Pin_2, B_Down);
      Pins.Read (Pins.Pin_3, B_Up);
```

```
      if not B_Down and not B_Up then
         B := Both;
      elsif not B_Down and B_Up then
         B := Down;
      elsif B_Down and not B_Up then
         B := Up;
      else
         B := None;
      end if;
   end Read_Buttons;

   -- interrupt handler:
   procedure Handle_Buttons is
      Buttons : Buttons_State;
   begin
      Read_Buttons (Buttons);

      Current_State := Transitions (Current_State, Buttons);

      Control_Motor;

      Utils.Waste_Some_Time;
      Pins.Clear_Interrupt (Pins.Pin_2);
      Pins.Clear_Interrupt (Pins.Pin_3);
   end Handle_Buttons;

   procedure Run is
   begin

      -- initialize the device
      Pins.Enable_Input (Pins.Pin_2, Pins.Pulled_Up);
      Pins.Enable_Input (Pins.Pin_3, Pins.Pulled_Up);
      Pins.Enable_Output (Pins.Pin_4);
      Pins.Enable_Output (Pins.Pin_5);
      Pins.Enable_Output (Pins.Pin_6);

      Current_State := Stop;
      Control_Motor;

      -- allow interrupts
      Utils.Enable_Interrupts;
      Pins.Enable_Interrupts (Pins.Pin_2);
      Pins.Enable_Interrupts (Pins.Pin_3);

      -- do nothing,
      -- the whole activity is driven by interrupts
      loop
         Utils.Wait_For_Interrupt;
      end loop;
   end Run;

end Program;
```

The delay introduced in procedure `Handle_Buttons` allows one to *debounce* interrupts that might arrive in a very fast sequence due to electric glitches when the button is physically pressed. This delay is effective, but is not a very optimal solution as it suspends execution of the interrupt handler - we will replace it with something more appropriate later on.

Now the program is completely driven by external events (buttons being pressed) and does not perform any activity unless *provoked* by the user. We will not argue that this will result in significant power consumption savings (it is a *fan* control system, after all), but for sure such a design can be easier to integrate with other additional functions if necessary.

Note that we have also refactored the original procedure `Control_Motor` - it was modified to use constant array `Outputs` instead of a longer `case` statement to decide which pins should be "high" or "low" in any given state of the system. The idea is the same as with the `Transitions` table, which allows to reduce the amount of code by expressing some of the program logic in terms of configuration data. Note that in the definition of the array we have used a subset (a subsequence) of the original type defining pin names. This was possible because the pins that are used to control the fan motor in our system form a contiguous subset of the original type. In other words, the following two definitions:

```
   type Pin_ID is ( Pin_2, Pin_3, Pin_4, Pin_5,
      Pin_6, Pin_7, Pin_8, Pin_9,
      Pin_10, Pin_11, Pin_12, Pin_13 );

   subtype Output_Pins is Pin_ID range Pin_4 .. Pin_6;
```

allow to refer to a *subset* of all pins (4 - 6) without introducing a new (and incompatible) set of names. Such a named subset can be used to define array index ranges just as the original type. This feature of the Ada type system is available for all discrete types.

In other words, in the following line of the `Control_Motor` procedure:

```
         Pins.Write (Pins.Pin_4, Outputs (Current_State, Pins.Pin_4));
```

the expression:

```
                                  Outputs (Current_State, Pins.Pin_4)
```

refers to the `Outputs` array to get the value that output pin 4 should have for the logical state denoted by parameter `Current_State`. This array contains proper configuration for all three output pins (4, 5 and 6) for each of the four logical states of the system. The constant aggregate was written in a compact form instead of a more verbose form with named value associations, as was done for `Transitions` - use your engineering judgment to decide if this shorter form is justified in this particular case.

It should be possible to restructure the `Read_Buttons` procedure in a similar way and implement the whole program on top of three constant arrays (input pin mapping, state transitions and output pin mapping) with a minimum of supporting code. It is not clear whether this would lead to better code readability, but it could be an interesting exercise in replacing code logic with configuration data. Such techniques are occasionally useful when implementing finite state machines, but should not be overused, as replacing program code with configuration data can make later debugging less intuitive.

Previous: Shared State, next: System Timer.
See also Table of Contents.