13. Interrupts

In all previous chapters we have been working with example programs that have a common structure: initialization phase followed by a loop, which managed the whole activity of the program. If the program was supposed to do nothing, then that *doing nothing* was implemented by means of a tight loop that wasted cycles; if instead the program was supposed to react to some external events like buttons being pressed, then again the tight loop was responsible for repeatedly testing the inputs. In both cases the program was active even when the idea was to wait.

This approach has the advantage of being very simple, but the price of this simplicity is in high energy consumption and, ironically, in more difficult integration of independent tasks. For example, combining the fan control system with a blinking LED would be difficult even though these functions are very simple when implemented in separation. Clearly, this design approach has its limitations.

Interrupts is a mechanism that allows decoupling functions that are responsible for handling separate events and are a low-level method of implementing event-driven designs. The idea is that instead of actively testing the environment (including the passage of time) for interesting conditions, interrupts allow separate pieces of code to *subscribe* to interesting events - then, when the given event takes place, like the button is pressed or the given timeout has elapsed, etc., the selected procedure is automatically invoked in order to handle it.

The ARM Cortex-M microcontrollers have very flexible support for interrupts and there are many ways to use them. As a matter of exercise we will try to do something very simple like reacting to the button being pressed (from the point of view of the microcontroller, this is visible as a change of state of the input pin), but first we need to explore the documentation to get some facts together.

Interrupts related to input pins can be configured at three different levels:

- globally, by means of enabling or disabling the delivery of interrupts (whatever their origin) at the processor level - in our case the procedures `Enable_Interrupts` as well as `Disable_Interrupts` that we have prepared in the previous chapter can be used for this,
- with the *Nested Vectored Interrupt Controller* (NVIC), which can enable or disable interrupts coming from individual peripherals as well and manage their priorities and other properties, and
- at the level of individual sources like single lines of the I/O peripheral.

This three-level management hierarchy is very flexible, but we will not need all of its potential and we will simplify some bits whenever possible. Still, it is worth to note that the frist two levels above (global and NVIC) are handled in a similar way in all Cortex-M microcontrollers, as this is part of the core ARM design - however, the configuration of individual peripheral interrupt sources tends to differ between microcontrollers.

We already have procedures `Enable_Interrupts` and `Disable_Interrupts` in the package `Utils`, but in order to configure the rest we will need to refer to some new registers. For the sake of example, let's focus on pin 3 on the Arduino connector - below are the relevant pieces of information that we will need to use, of course different for each microcontroller.

Arduino M0

For Arduino M0 we have to connect the following dots scattered across several documentation chapters:

- pin 3 of the Arduino connector is handled by line 9 of Port A (we already know this),
- from the *I/O Multiplexing* chapter of the documentation we can see that I/O pins are shared with alternative peripherals - one of the alternatives is the *External Interrupt Controller* (EIC), which has the capability to generate interrupts based on changes on the I/O pins,
- in order to activate the alternative peripheral, the I/O has to be configured in the multiplexing mode and the correct alternative has to be selected in the multiplexer,
- in the *PM - Power Manager* chapter we can see that the EIC module has a bus clock signal (CLK_EIC_APB) that is enabled by default - we will rely on this to simplify our setup procedure,
- the EIC module has several controlling registers for enabling its operation and for configuring the kind of events that it is expected to react to - we will ask for edge detection in both directions,
- we can also see that in order to detect edges on I/O pins the EIC module needs to have additional clock signal (GCLK_EIC) enabled - this can be done in the *Generic Clock Controller* (GCLK) and we will count on the fact that one of several existing clock generators (GCLKGEN0) is by default active, so we can configure it as a source for GCLK_EIC, this

configuration can be done with register GCLK_CLKCTRL),

- the *Nested Vectored Interrupt Controller* has slot 4 assigned for interrupts generated by EIC - this slot has to be activated by setting appropriate bit in register ISER.

Note that according to the I/O multiplexing table, several I/O pins are managed by the same external interrupt line (EXTINTx). This might cause some problems as will see at the end of this chapter.

The above looks like a very complex interrupt management system, and in fact it is, as a consequence of the very flexible clocking system in the microcontroller in Arduino M0. In order to handle all these details our infrastructure needs to be extended as well:

- file `flash.ld` needs new symbols for registers ISER (for enabling EIC interrupts at the NVIC level), GCLK_CLKCTRL (for configuring clock for EIC), EIC_CTRL (for enabling EIC), EIC_INTENTSET (for enabling individual EXTINT lines), also EIC_CONFIG1 (for selecting the sensitivity of the I/O pins), EIC_INTFLAG (for marking the interrupts as already handled) and a set of additional GPIOA registers for configuring pin multiplexing,
- package `Registers` needs to have new register definitions,
- package `Pins` needs new procedures for enabling, disabling and clearing of interrupts for each pin separately.

Note that GCLK_CLKCTRL is a 16-bit register - in order to ensure proper read/write accesses for this register we will extend the package `Registers` with this definition:

```
type Half_Word is mod 2**16;
```

and of course we will use this type to instruct the compiler about the register size:

```
GCLK_CLKCTRL : Half_Word;
pragma Volatile (GCLK_CLKCTRL);
pragma Import (C, GCLK_CLKCTRL, "GCLK_CLKCTRL");
```

We also need additional procedures for enabling/disabling interrupts for individual pins and we will put them in package `Pins`, together with other pin-controlling code.

The procedure `Enable_Interrupts` can have the following structure:

```
procedure Enable_Interrupts (Pin : in Pin_ID) is
begin
   -- enable EIC at the level of NVIC:
   Registers.ISER := Registers.Bit_4;

   -- enable GCLK_EIC from GCLKGEN0:
   Registers.GCLK_CLKCTRL := 2#0100_0000_0000_0101#;

   -- enable EIC:
   Registers.EIC_CTRL := Registers.Bit_1;

   case Pin is
      -- ...

      when Pin_3 => -- PA09 -> EXTINT9
         -- enable interrupts at the level of single EXTINT line:
         Registers.EIC_INTENTSET := Registers.EIC_INTENTSET
            or Registers.Bit_9;

         -- react to both rising and falling edges:
         Registers.EIC_CONFIG1 := (Registers.EIC_CONFIG1
           and 2#1111_1111_1111_1111_1111_1111_0000_1111#)
            or 2#0000_0000_0000_0000_0000_0000_0011_0000#;

         -- enable and configure multiplexing
         -- to activate alternative peripheral (EIC) on this pin:
         Registers.GPIOA_PMUX4 := Registers.GPIOA_PMUX4
           and 2#0000_1111#;
         Registers.GPIOA_PINCFG9 := Registers.GPIOA_PINCFG9
           or Registers.Bit_0;

      -- ...
   end case;
end Enable_Interrupts;
```

We might also need a complementary procedure for disabling interrupts for the given pin - this procedure will be simpler, since there is no need to roll-back everything, it is enough to switch back from alternative peripheral to regular I/O pin operation:

```
procedure Disable_Interrupts (Pin : in Pin_ID) is
```

```
begin
   case Pin is
      -- ...

      when Pin_3 =>
         Registers.GPIOA_PINCFG9 := Registers.GPIOA_PINCFG9
           and (not Registers.Bit_0);

      -- ...
   end case;
end Disable_Interrupts;
```

The last operation that will be needed when working with interrupts is `Clear_Interrupts`, which should clear the pending flag once we consider the interrupt to be already handled. According to the documentation, in order to clear the pending flag it is enough to write appropriate bit in the EIC_INTFLAG register:

```
procedure Clear_Interrupt (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 =>
         Registers.EIC_INTFLAG := Registers.Bit_9;

      -- ...
   end case;
end Clear_Interrupt;
```

Of course, these procedures have to be completed for all other pins if we want to use them with interrupts.

Arduino Due

For Arduino Due we need to know that:

- pin 3 of the Arduino connector is handled by line 28 of PIOC (we have already relied on this),
- in chapter *Peripheral Identifiers* of the Atmel documentation we can find that PIOC has identifier 13 (we have already found it in previous chapters, but let's restate it for completeness) and that peripheral identifiers are also used to control interrupts with the NVIC (chapter *Exception types* confirms this as well),
- chapter *Nested Vectored Interrupt Controller* shows that interrupt 13 can be enabled/disabled by actions on registers ISER0 and ICER0,
- chapter *Input Edge/Level Interrupt* states that interrupts from individual PIO lines are enabled and disabled by actions on registers PIO_IER and PIO_IDR, and the same chapter also describes the role of PIO_ISR (Interrupt Status) register: the interrupt is cleared only when this register is read.

Note that from the point of view of NVIC, *all* lines of PIOC are managed together as interrupt 13. This means that in order to enable interrupts on a single line we need to enable it at both the peripheral (PIO) and NVIC, but special care should be taken when interrupts are disabled, because disabling interrupt 13 at the level of NVIC will block all other, perhaps unrelated lines from the same peripheral.

All this looks like a lot of stuff to manage, but fortunately we have most of the infrastructure already in place. We will need the following extensions:

- file `flash.ld` needs new symbols to define registers ISER0 and PIO_IER with PIO_IDR and PIO_ISR for peripherals handling the pins that we want to manage,
- package `Registers` needs to have new register definitions,
- package `Pins` needs new procedures for enabling, disabling and clearing of interrupts for each pin separately.

In order to simplify our API we can uniformly enable interrupts with NVIC whenever some pin is configured as input and we can just ignore the disabling part. That is, the procedure `Enable_Input` can be extended like here:

```
procedure Enable_Input (Pin : in Pin_ID; Mode : in Input_Mode) is
begin
   case Pin is
      -- ...

      when Pin_3 =>
         Registers.PMC_PCER0 := Registers.Bit_13;
         Registers.ISER0 := Registers.Bit_13;      -- here

         -- ...

      -- ...
   end case;
end Enable_Input;
```

Thanks to this, whenever some pin is configured as input, the NVIC is already prepared to handle interrupts from the given line (note that the interrupts will not be generated unless this is configured at the PIO level for that line).

We also need additional procedures for enabling/disabling interrupts for individual pins - these can be implemented according to this pattern:

```
procedure Enable_Interrupts (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 =>
         Registers.PIOC_IER := Registers.Bit_28;

      -- ...
   end case;
end Enable_Interrupts;

procedure Disable_Interrupts (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 =>
         Registers.PIOC_IDR := Registers.Bit_28;

      -- ...
   end case;
end Disable_Interrupts;

procedure Clear_Interrupt (Pin : in Pin_ID) is
   Dummy : Registers.Word;
begin
   case Pin is
      -- ...

      when Pin_3 =>
         Dummy := Registers.PIOC_ISR;

      -- ...
   end case;
end Clear_Interrupt;
```

Of course, these procedures have to be completed for all other pins if we want to use them with interrupts.

STM32 Nucleo-32

For Nucleo-32 we need to know that:

- pin 3 of the Arduino connector is handled by line PB0, which is managed by GPIOB (we have already relied on this),
- in chapter *External and internal interrupt/event line mapping* of the STM32 documentation we can see that PB0 can be multiplexed by EXTI0, which is a way for the multicontroller to handle I/O pin-related events - the multiplexing is configured with the help of register SYSCFG_EXTICR1,
- the SYSCFG module itself requires a clock signal to be delivered in order to work properly and this is done by means of the RCC_APB2ENR register (bit 0),
- EXTI0 has its slot, together with EXTI1, as the 6th interrupt above standard ones in the vector table - that is, it is handled by IRQ5 line of the NVIC controller,
- chapter *Nested Vectored Interrupt Controller* in the ARM reference shows that such interrupts can be enabled/disabled by actions on registers ISER and ICER,
- chapter *EXTI registers* states that interrupts from individual EXTI lines are enabled and disabled by actions on registers EXTI_IMR, EXTI_RTSR and EXTI_FTSR, and that the EXTI_PR, the pending register, accounts for events that have not yet been handled - writing 1 to the selected bit clears the pending state.

Note that from the point of view of NVIC, different I/O lines are handled by different EXTI lines, but the multiplexing performed by EXTI means that not all I/O pins can be used with interrupts at the same time, see notes at the end of this chapter. In any case, interrupts are routed at two levels and have to be enabled at both the EXTI and NVIC. Also, special care should be taken when interrupts are disabled, because disabling interrupts at the level of NVIC can block all other, perhaps unrelated EXTI lines.

All this looks like a lot of stuff to manage, but fortunately we have most of the infrastructure already in place. We will need the following extensions:

- file `flash.ld` needs new symbols to define registers RCC_APB2ENR, ISER as well as SYSCFG_EXTICR1 and EXTI_IMR, EXTI_RTSR, EXTI_FTSR and EXTI_PR, so that we can control interrupt configuration at the level of NVIC, EXTI multiplexing and EXTI interrupt masking, respectively,

- package `Registers` needs to have new register definitions to reflect new symbols from the linker script,
- package `Pins` needs new procedures for enabling, disabling and clearing of interrupts for each pin separately.

In order to simplify our API we can enable interrupts with NVIC whenever some pin is configured for use with interrupts and we can just ignore the disabling part (that is, we will not need the ICER register).

We need additional procedures for enabling/disabling interrupts for individual pins - these can be implemented according to this pattern:

```
procedure Enable_Interrupts (Pin : in Pin_ID) is
begin
   Registers.RCC_APB2ENR := Registers.RCC_APB2ENR
      or Registers.Bit_0;

   case Pin is
      -- ...

      when Pin_3 => -- PB0 in EXTI0
         Registers.SYSCFG_EXTICR1 := (Registers.SYSCFG_EXTICR1
            and 2#1111_1111_1111_1111_1111_1111_1111_0000#)
             or 2#0000_0000_0000_0000_0000_0000_0000_0001#;
         Registers.EXTI_IMR := Registers.EXTI_IMR
           or Registers.Bit_0;
         Registers.EXTI_RTSR := Registers.EXTI_RTSR
           or Registers.Bit_0;
         Registers.EXTI_FTSR := Registers.EXTI_FTSR
           or Registers.Bit_0;

         Registers.ISER := Registers.Bit_5;

      -- ...
   end case;
end Enable_Interrupts;

procedure Disable_Interrupts (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 => -- PB0 in EXTI0
         Registers.EXTI_IMR := Registers.EXTI_IMR
           and 2#1111_1111_1111_1111_1111_1111_1111_1110#;

      -- ...
   end case;
end Disable_Interrupts;

procedure Clear_Interrupt (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 => -- PB0 in EXTI0
         Registers.EXTI_PR := Registers.Bit_0;

      -- ...
   end case;
end Clear_Interrupt;
```

Of course, these procedures have to be completed for all other pins if we want to use them with interrupts.

STM32 Nucleo-144

For Nucleo-144 we need to know that:

- pin 3 of the Arduino connector is handled by line PE13, which is managed by GPIOE (we have already relied on this),
- in chapter *External and internal interrupt/event line mapping* of the STM32 documentation we can see that PE13 can be multiplexed by EXTI13, which is a way for the multicontroller to handle I/O pin-related events - the multiplexing is configured with the help of register SYSCFG_EXTICR4,
- the SYSCFG module itself requires a clock signal to be delivered in order to work properly and this is done by means of the RCC_APB2ENR register (bit 14),
- EXTI13 has its slot, together with the whole group EXTI10-EXTI15, as the interrupt 40 above standard ones in the vector table - that is, it is handled by IRQ40 line of the NVIC controller,
- chapter *Nested Vectored Interrupt Controller* in the ARM reference shows that such interrupts can be enabled/disabled by actions on registers ISER and ICER; what is not clearly explained is that a single register can control at most 32 interrupts, whereas we are already above that range with the slot for IRQ40 - the solution that we have already seen with Arduino Due is to have a whole sequence of registers that together form an array of controlling bits longer than 32, so that ISER0 might take care of interrupts 0-31, ISER1 handles interrupts in the range 32-63, and so on,

depending on the number of interrupt lines foreseen by the chip vendor, and in this scheme interrupt line 40 is managed by bit 8 in ISER1 (similar reasoning applies to the ICER register, as well),

- chapter *EXTI registers* states that interrupts from individual EXTI lines are enabled and disabled by actions on registers EXTI_IMR, EXTI_RTSR and EXTI_FTSR, and that the EXTI_PR, the pending register, accounts for events that have not yet been handled - writing 1 to the selected bit clears the pending state.

Note that from the point of view of NVIC, different I/O lines are handled by different EXTI lines, but the multiplexing performed by EXTI means that not all I/O pins can be used with interrupts at the same time, see notes at the end of this chapter. In any case, interrupts are routed at two levels and have to be enabled at both the EXTI and NVIC. Also, special care should be taken when interrupts are disabled, because disabling interrupts at the level of NVIC can block all other, perhaps unrelated EXTI lines.

All this looks like a lot of stuff to manage, but fortunately we have most of the infrastructure already in place. We will need the following extensions:

- file `flash.ld` needs new symbols to define registers RCC_APB2ENR, ISER0, ISER1, SYSCFG_EXTICR4, EXTI_IMR, EXTI_RTSR, EXTI_FTSR and EXTI_PR so that we can control interrupt configuration at the level of NVIC, EXTI multiplexing and EXTI interrupt masking, respectively,
- package `Registers` needs to have new register definitions to reflect new symbols from the linker script,
- package `Pins` needs new procedures for enabling, disabling and clearing of interrupts for each pin separately.

In order to simplify our API we can enable interrupts with NVIC whenever some pin is configured for use with interrupts and we can just ignore the disabling part (that is, we will not need the ICERx registers).

We need additional procedures for enabling/disabling interrupts for individual pins - these can be implemented according to this pattern:

```
procedure Enable_Interrupts (Pin : in Pin_ID) is
begin
   Registers.RCC_APB2ENR := Registers.RCC_APB2ENR
      or Registers.Bit_14;

   case Pin is
      -- ...

      when Pin_3 => -- PE13 in EXTI13
         Registers.SYSCFG_EXTICR4 := (Registers.SYSCFG_EXTICR4
            and 2#1111_1111_1111_1111_1111_1111_0000_1111#)
             or 2#0000_0000_0000_0000_0000_0000_0100_0000#;
         Registers.EXTI_IMR := Registers.EXTI_IMR
           or Registers.Bit_13;
         Registers.EXTI_RTSR := Registers.EXTI_RTSR
           or Registers.Bit_13;
         Registers.EXTI_FTSR := Registers.EXTI_FTSR
           or Registers.Bit_13;

         Registers.ISER1 := Registers.Bit_8;

      -- ...
   end case;
end Enable_Interrupts;

procedure Disable_Interrupts (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 => -- PE13 in EXTI13
         Registers.EXTI_IMR := Registers.EXTI_IMR
           and 2#1111_1111_1111_1111_1101_1111_1111_1111#;

      -- ...
   end case;
end Disable_Interrupts;

procedure Clear_Interrupt (Pin : in Pin_ID) is
begin
   case Pin is
      -- ...

      when Pin_3 => -- PE13 in EXTI13
         Registers.EXTI_PR := Registers.Bit_13;

      -- ...
   end case;
end Clear_Interrupt;
```

Of course, these procedures have to be completed for all other pins if we want to use them with interrupts.

Now, with all these supporting procedures in place, we can attempt to write a simple program that uses interrupts to react on state changes on pin 3 - by, for example, setting pin 4 to the same state. This is the complete program that does it:

```
with Pins;
with Utils;

package body Program is

   procedure Button_Pressed is
      State : Boolean;
   begin
      -- copy state from pin 3 to pin 4
      Pins.Read (Pins.Pin_3, State);
      Pins.Write (Pins.Pin_4, State);

      Pins.Clear_Interrupt (Pins.Pin_3);
   end Button_Pressed;

   procedure Run is
   begin
      -- initialize the device
      Pins.Enable_Input (Pins.Pin_3, Pins.Pulled_Up);
      Pins.Enable_Output (Pins.Pin_4);

      -- allow interrupts
      Utils.Enable_Interrupts;
      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;
```

This program is remarkably short, but there are some interesting things happening here. The procedure `Run` is the one that is executed right after restart of the device - just as before. It starts by setting pin 3 for input and pin 4 for output and then it enables interrupts, both at the global level and at the level of pin 3. Remember that NVIC was automatically configured from `Enable_Input`, so we have complete configuration done in just few lines of code. After that, the `Run` procedure has nothing else to do, so it enters the infinite loop.

Note that previously we have used the loop of this form:

```
      loop
         null;
      end loop;
```

This was correct in the sense that prevented the procedure `Run` from exiting, but was wasteful as the loop was forced to spin at the highest possible speed, thus wasting cycles and energy. By introducing the invocation of `Wait_For_Interrupt`, we allow the processor to suspend execution until some interrupt arrives - during that time the processor consumes a lot less power and this is a recommended practice for event-driven designs. Note, however, that the ARM architecture allows implementations to treat this functionality only as an optional feature - that is, the processor is *not* required to actually suspend execution and is allowed to continue just as with the null statement (at the level of processor, the `WFI` instruction is allowed to execute as `NOP`). This is why the recommended practice is to always call `WFI` in the loop:

```
      loop
         Utils.Wait_For_Interrupt;
      end loop;
```

This way the code works correctly independently on how this feature is implemented at the microcontroller level.

The `Button_Pressed` procedure simply copies the state of pin 3 to pin 4 *and returns*, which means that it was prepared for a single-shot execution, not for continuous control - this is exactly what we need, since we expect this procedure to be called automatically whenever the state of pin 3 changes. Note that this procedure *clears* the interrupt after handling it, to prevent the immediate invocation for the same interrupt that would be otherwise still active at its source.

You should notice that this procedure is not connected in any way to the `Run` procedure at the level of source code. In fact, both procedures are *handlers*, which are invoked to handle different events: the `Run` procedure handles the reset event and the `Button_Pressed` procedure is supposed to handle the state change of pin 3. Both procedures are wired at the level of the linker script, taking into account the external names that are assigned to these procedures.

This is the specification of package `Program`:

```
package Program is

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

   procedure Button_Pressed;
   pragma Export (C, Button_Pressed, "button_pressed");

end Program;
```

The complete wiring of these procedures can be found in the linker script, in the section that lays out the initial words of the final binary image:

Arduino M0

For Arduino M0, the part of the linker script that lays out the vector table and takes interrupt handlers into account looks like:

```
    .vectors 0x00004000 :
    {
        LONG(0x20008000)
        LONG(run + 1)
        LONG(0)                   /* 0x08 -- nmi                  */
        LONG(0)                   /* 0x0c -- hard fault           */
        LONG(0)                   /* 0x10                         */
        LONG(0)                   /* 0x14                         */
        LONG(0)                   /* 0x18                         */
        LONG(0)                   /* 0x1c                         */
        LONG(0)                   /* 0x20                         */
        LONG(0)                   /* 0x24                         */
        LONG(0)                   /* 0x28                         */
        LONG(0)                   /* 0x2c -- svcall               */
        LONG(0)                   /* 0x30                         */
        LONG(0)                   /* 0x34                         */
        LONG(0)                   /* 0x38 -- pendsv               */
        LONG(0)                   /* 0x3c -- systick              */
        LONG(0)                   /* 0x40 -- interrupt 0          */
        LONG(0)                   /* 0x44 -- interrupt 1          */
        LONG(0)                   /* 0x48 -- interrupt 2          */
        LONG(0)                   /* 0x4c -- interrupt 3          */
        LONG(button_pressed + 1) /* 0x50 -- EIC                   */
        LONG(0)                   /* 0x54 -- interrupt 5          */
        LONG(0)                   /* 0x58 -- interrupt 6          */
        LONG(0)                   /* 0x5c -- interrupt 7          */
        LONG(0)                   /* 0x60 -- interrupt 8          */
        LONG(0)                   /* 0x64 -- interrupt 9          */
        LONG(0)                   /* 0x68 -- interrupt 10         */
        LONG(0)                   /* 0x6c -- interrupt 11         */
        LONG(0)                   /* 0x70 -- interrupt 12         */
        LONG(0)                   /* 0x74 -- interrupt 13         */
        LONG(0)                   /* 0x78 -- interrupt 14         */
        LONG(0)                   /* 0x7c -- interrupt 15         */
        LONG(0)                   /* 0x80 -- interrupt 16         */
        LONG(0)                   /* 0x84 -- interrupt 17         */
        LONG(0)                   /* 0x88 -- interrupt 18         */
        LONG(0)                   /* 0x8c -- interrupt 19         */
        LONG(0)                   /* 0x90 -- interrupt 20         */
        LONG(0)                   /* 0x94 -- interrupt 21         */
        LONG(0)                   /* 0x98 -- interrupt 22         */
        LONG(0)                   /* 0x9c -- interrupt 23         */
        LONG(0)                   /* 0xa0 -- interrupt 24         */
        LONG(0)                   /* 0xa4 -- interrupt 25         */
        LONG(0)                   /* 0xa8 -- interrupt 26         */
        LONG(0)                   /* 0xac -- interrupt 27         */
        FILL(0)
    }
```

We already know that interrupts coming from EIC have dedicated slot 4, so the procedure handling these interrupts should be wired at offset 0x50 of the table (remember that first bit in all vectors should be set, this is why symbol values are incremented). As you can see, there is only one handler for all EIC interrupts, so if multiple I/O pins are generating them, the handler has to be written in a way that allows it to distinguish what is the actual source.

Arduino Due

For Arduino Due, the part of the linker script that lays out the vector table and takes interrupt handlers into account looks like:

```
    .vectors 0x00080000 :
    {
        LONG(0x20088000)          /* 0x00 -- initial stack pointer */
        LONG(run + 1)             /* 0x04 -- reset                 */
        LONG(0)                   /* 0x08                          */
```

```
        LONG(0)                 /* 0x0c -- hard fault           */
        LONG(0)                 /* 0x10 -- memory mgmt fault    */
        LONG(0)                 /* 0x14 -- bus fault            */
        LONG(0)                 /* 0x18 -- usage fault          */
        LONG(0)                 /* 0x1c                         */
        LONG(0)                 /* 0x20                         */
        LONG(0)                 /* 0x24                         */
        LONG(0)                 /* 0x28                         */
        LONG(0)                 /* 0x2c -- svcall               */
        LONG(0)                 /* 0x30 -- reserved for debug   */
        LONG(0)                 /* 0x34                         */
        LONG(0)                 /* 0x38 -- pendsv               */
        LONG(0)                 /* 0x3c -- systick              */
        LONG(0)                 /* 0x40 -- interrupt 0          */
        LONG(0)                 /* 0x44 -- interrupt 1          */
        LONG(0)                 /* 0x48 -- interrupt 2          */
        LONG(0)                 /* 0x4c -- interrupt 3          */
        LONG(0)                 /* 0x50 -- interrupt 4          */
        LONG(0)                 /* 0x54 -- interrupt 5          */
        LONG(0)                 /* 0x58 -- interrupt 6          */
        LONG(0)                 /* 0x5c -- interrupt 7          */
        LONG(0)                 /* 0x60 -- interrupt 8          */
        LONG(0)                 /* 0x64 -- interrupt 9          */
        LONG(0)                 /* 0x68 -- interrupt 10         */
        LONG(0)                 /* 0x6c -- interrupt 11         */
        LONG(0)                 /* 0x70 -- interrupt 12 (PIOB)  */
        LONG(button_pressed + 1) /* 0x74 -- interrupt 13 (PIOC)  */
        LONG(0)                 /* 0x78 -- interrupt 14 (PIOD)  */
        LONG(0)                 /* 0x7c -- interrupt 15         */
        LONG(0)                 /* 0x80 -- interrupt 16         */
        LONG(0)                 /* 0x84 -- interrupt 17         */
        LONG(0)                 /* 0x88 -- interrupt 18         */
        LONG(0)                 /* 0x8c -- interrupt 19         */
        LONG(0)                 /* 0x90 -- interrupt 20         */
        LONG(0)                 /* 0x94 -- interrupt 21         */
        LONG(0)                 /* 0x98 -- interrupt 22         */
        LONG(0)                 /* 0x9c -- interrupt 23         */
        LONG(0)                 /* 0xa0 -- interrupt 24         */
        LONG(0)                 /* 0xa4 -- interrupt 25         */
        LONG(0)                 /* 0xa8 -- interrupt 26         */
        LONG(0)                 /* 0xac -- interrupt 27         */
        LONG(0)                 /* 0xb0 -- interrupt 28         */
        LONG(0)                 /* 0xb4 -- interrupt 29         */
        FILL(0)
    }
```

We already know that interrupts coming from PIOC have number 13, so the procedure handling these interrupts should be wired at offset 0x74 of the table (remember that first bit in all vectors should be set, this is why symbol values are incremented). This is what binds all parts of our program together.

STM32 Nucleo-32

For Nucleo-32, the part of the linker script that lays out the vector table and takes interrupt handlers into account looks like:

```
    .vectors 0x08000000 :
    {
        LONG(0x20001000)        /* 0x00 -- initial stack pointer */
        LONG(run + 1)           /* 0x04 -- reset                */
        LONG(0)                 /* 0x08 -- nmi                  */
        LONG(0)                 /* 0x0c -- hard fault           */
        LONG(0)                 /* 0x10                         */
        LONG(0)                 /* 0x14                         */
        LONG(0)                 /* 0x18                         */
        LONG(0)                 /* 0x1c                         */
        LONG(0)                 /* 0x20                         */
        LONG(0)                 /* 0x24                         */
        LONG(0)                 /* 0x28                         */
        LONG(0)                 /* 0x2c -- svcall               */
        LONG(0)                 /* 0x30                         */
        LONG(0)                 /* 0x34                         */
        LONG(0)                 /* 0x38 -- pendsv               */
        LONG(0)                 /* 0x3c -- systick              */
        LONG(0)                 /* 0x40 -- interrupt 0          */
        LONG(0)                 /* 0x44 -- interrupt 1          */
        LONG(0)                 /* 0x48 -- interrupt 2          */
        LONG(0)                 /* 0x4c -- interrupt 3          */
        LONG(0)                 /* 0x50 -- interrupt 4          */
        LONG(button_pressed + 1) /* 0x54 -- EXTI0_1             */
        LONG(0)                 /* 0x58 -- interrupt 6          */
        LONG(0)                 /* 0x5c -- interrupt 7          */
        LONG(0)                 /* 0x60 -- interrupt 8          */
        LONG(0)                 /* 0x64 -- interrupt 9          */
        LONG(0)                 /* 0x68 -- interrupt 10         */
        LONG(0)                 /* 0x6c -- interrupt 11         */
        LONG(0)                 /* 0x70 -- interrupt 12         */
        LONG(0)                 /* 0x74 -- interrupt 13         */
```

```
        LONG(0)                 /* 0x78 -- interrupt 14         */
        LONG(0)                 /* 0x7c -- interrupt 15         */
        LONG(0)                 /* 0x80 -- interrupt 16         */
        LONG(0)                 /* 0x84 -- interrupt 17         */
        LONG(0)                 /* 0x88 -- interrupt 18         */
        LONG(0)                 /* 0x8c -- interrupt 19         */
        LONG(0)                 /* 0x90 -- interrupt 20         */
        LONG(0)                 /* 0x94 -- interrupt 21         */
        LONG(0)                 /* 0x98 -- interrupt 22         */
        LONG(0)                 /* 0x9c -- interrupt 23         */
        LONG(0)                 /* 0xa0 -- interrupt 24         */
        LONG(0)                 /* 0xa4 -- interrupt 25         */
        LONG(0)                 /* 0xa8 -- interrupt 26         */
        LONG(0)                 /* 0xac -- interrupt 27         */
        LONG(0)                 /* 0xb0 -- interrupt 28         */
        LONG(0)                 /* 0xb4 -- interrupt 29         */
        LONG(0)                 /* 0xb8 -- interrupt 30         */
        LONG(0)                 /* 0xbc -- interrupt 31         */
        FILL(0)
    }
```

STM32 Nucleo-144

For Nucleo-144, the part of the linker script that lays out the vector table and takes interrupt handlers into account looks like (deliberately shown in its entirety - note how careful we have to be in the proper placing of each particular handler):

```
    .vectors 0x08000000 :
    {
        LONG(0x20010000)        /* 0x000 -- initial stack pointer */
        LONG(run + 1)           /* 0x004 -- reset                 */
        LONG(0)                 /* 0x008 -- nmi                   */
        LONG(0)                 /* 0x00c -- hard fault            */
        LONG(0)                 /* 0x010                          */
        LONG(0)                 /* 0x014                          */
        LONG(0)                 /* 0x018                          */
        LONG(0)                 /* 0x01c                          */
        LONG(0)                 /* 0x020                          */
        LONG(0)                 /* 0x024                          */
        LONG(0)                 /* 0x028                          */
        LONG(0)                 /* 0x02c -- svcall                */
        LONG(0)                 /* 0x030                          */
        LONG(0)                 /* 0x034                          */
        LONG(0)                 /* 0x038 -- pendsv                */
        LONG(0)                 /* 0x03c -- systick               */
        LONG(0)                 /* 0x040 -- interrupt 0           */
        LONG(0)                 /* 0x044 -- interrupt 1           */
        LONG(0)                 /* 0x048 -- interrupt 2           */
        LONG(0)                 /* 0x04c -- interrupt 3           */
        LONG(0)                 /* 0x050 -- interrupt 4           */
        LONG(0)                 /* 0x054 -- interrupt 5           */
        LONG(0)                 /* 0x058 -- interrupt 6           */
        LONG(0)                 /* 0x05c -- interrupt 7           */
        LONG(0)                 /* 0x060 -- interrupt 8           */
        LONG(0)                 /* 0x064 -- interrupt 9           */
        LONG(0)                 /* 0x068 -- interrupt 10          */
        LONG(0)                 /* 0x06c -- interrupt 11          */
        LONG(0)                 /* 0x070 -- interrupt 12          */
        LONG(0)                 /* 0x074 -- interrupt 13          */
        LONG(0)                 /* 0x078 -- interrupt 14          */
        LONG(0)                 /* 0x07c -- interrupt 15          */
        LONG(0)                 /* 0x080 -- interrupt 16          */
        LONG(0)                 /* 0x084 -- interrupt 17          */
        LONG(0)                 /* 0x088 -- interrupt 18          */
        LONG(0)                 /* 0x08c -- interrupt 19          */
        LONG(0)                 /* 0x090 -- interrupt 20          */
        LONG(0)                 /* 0x094 -- interrupt 21          */
        LONG(0)                 /* 0x098 -- interrupt 22          */
        LONG(0)                 /* 0x09c -- interrupt 23          */
        LONG(0)                 /* 0x0a0 -- interrupt 24          */
        LONG(0)                 /* 0x0a4 -- interrupt 25          */
        LONG(0)                 /* 0x0a8 -- interrupt 26          */
        LONG(0)                 /* 0x0ac -- interrupt 27          */
        LONG(0)                 /* 0x0b0 -- interrupt 28          */
        LONG(0)                 /* 0x0b4 -- interrupt 29          */
        LONG(0)                 /* 0x0b8 -- interrupt 30          */
        LONG(0)                 /* 0x0bc -- interrupt 31          */
        LONG(0)                 /* 0x0c0 -- interrupt 32          */
        LONG(0)                 /* 0x0c4 -- interrupt 33          */
        LONG(0)                 /* 0x0c8 -- interrupt 34          */
        LONG(0)                 /* 0x0cc -- interrupt 35          */
        LONG(0)                 /* 0x0d0 -- interrupt 36          */
        LONG(0)                 /* 0x0d4 -- interrupt 37          */
        LONG(0)                 /* 0x0d8 -- interrupt 38          */
        LONG(0)                 /* 0x0dc -- interrupt 39          */
        LONG(button_pressed + 1) /* 0x0e0 -- EXTI15_10            */
        LONG(0)                 /* 0x0e4 -- interrupt 41          */
        LONG(0)                 /* 0x0e8 -- interrupt 42          */
```

```
        LONG(0)                 /* 0x0ec -- interrupt 43         */
        LONG(0)                 /* 0x0f0 -- interrupt 44         */
        LONG(0)                 /* 0x0f4 -- interrupt 45         */
        LONG(0)                 /* 0x0f8 -- interrupt 46         */
        LONG(0)                 /* 0x0fc -- interrupt 47         */
        LONG(0)                 /* 0x100 -- interrupt 48         */
        LONG(0)                 /* 0x104 -- interrupt 49         */
        LONG(0)                 /* 0x108 -- interrupt 50         */
        LONG(0)                 /* 0x10c -- interrupt 51         */
        LONG(0)                 /* 0x110 -- interrupt 52         */
        LONG(0)                 /* 0x114 -- interrupt 53         */
        LONG(0)                 /* 0x118 -- interrupt 54         */
        LONG(0)                 /* 0x11c -- interrupt 55         */
        LONG(0)                 /* 0x120 -- interrupt 56         */
        LONG(0)                 /* 0x124 -- interrupt 57         */
        LONG(0)                 /* 0x128 -- interrupt 58         */
        LONG(0)                 /* 0x12c -- interrupt 59         */
        LONG(0)                 /* 0x130 -- interrupt 60         */
        LONG(0)                 /* 0x134 -- interrupt 61         */
        LONG(0)                 /* 0x138 -- interrupt 62         */
        LONG(0)                 /* 0x13c -- interrupt 63         */
        LONG(0)                 /* 0x140 -- interrupt 64         */
        LONG(0)                 /* 0x144 -- interrupt 65         */
        LONG(0)                 /* 0x148 -- interrupt 66         */
        LONG(0)                 /* 0x14c -- interrupt 67         */
        LONG(0)                 /* 0x150 -- interrupt 68         */
        LONG(0)                 /* 0x154 -- interrupt 69         */
        LONG(0)                 /* 0x158 -- interrupt 70         */
        LONG(0)                 /* 0x15c -- interrupt 71         */
        LONG(0)                 /* 0x160 -- interrupt 72         */
        LONG(0)                 /* 0x164 -- interrupt 73         */
        LONG(0)                 /* 0x168 -- interrupt 74         */
        LONG(0)                 /* 0x16c -- interrupt 75         */
        LONG(0)                 /* 0x170 -- interrupt 76         */
        LONG(0)                 /* 0x174 -- interrupt 77         */
        LONG(0)                 /* 0x178 -- interrupt 78         */
        LONG(0)                 /* 0x17c -- interrupt 79         */
        LONG(0)                 /* 0x180 -- interrupt 80         */
        LONG(0)                 /* 0x184 -- interrupt 81         */
        LONG(0)                 /* 0x188 -- interrupt 82         */
        LONG(0)                 /* 0x18c -- interrupt 83         */
        LONG(0)                 /* 0x190 -- interrupt 84         */
        LONG(0)                 /* 0x194 -- interrupt 85         */
        LONG(0)                 /* 0x198 -- interrupt 86         */
        LONG(0)                 /* 0x19c -- interrupt 87         */
        LONG(0)                 /* 0x1a0 -- interrupt 88         */
        LONG(0)                 /* 0x1a4 -- interrupt 89         */
        LONG(0)                 /* 0x1a8 -- interrupt 90         */
        LONG(0)                 /* 0x1ac -- interrupt 91         */
        LONG(0)                 /* 0x1b0 -- interrupt 92         */
        LONG(0)                 /* 0x1b4 -- interrupt 93         */
        LONG(0)                 /* 0x1b8 -- interrupt 94         */
        LONG(0)                 /* 0x1bc -- interrupt 95         */
        LONG(0)                 /* 0x1c0 -- interrupt 96         */
        LONG(0)                 /* 0x1c4 -- interrupt 97         */
        FILL(0)
    }
```

Note that before we only needed the first two entries (initial stack pointer and the reset handler), but here the complete vector table was expanded, appropriately for each board. What is important in these expanded vector tables is that the initial slots are standardized and are responsible for managing the same kinds of exceptions, whereas slots above *SysTick* are implementation-defined and indeed we can see that there are no common rules even for such basic things like interrupts driven by I/O lines.

Another interesting observation is that both procedures `Run` and `Button_Pressed` execute in a way that is somewhat unrelated to each other just as multiple threads can have unrelated execution in a higher-level program. We will discuss it again in later chapters.

Final notes:

One of the things that are worth observing at this stage is that our procedures in the `Pins` package became uncomfortably long - especially if we the take into account that their structure is very repeatable, as each pin is handled in the same way, only differing by register names and bit patterns. This is similar to our first version of finite state machines, which also involved lengthy `case`/`when` ladders - and can be simplified in the same way. It is possible to create a constant structure (very likely an array, indexed by pin names) with the configuration data that changes between pins and then have much shorter procedures that pick values from that array in order to set proper bit patterns in proper registers, depending on the requested pin. Such solution will be very easy to extend for more pins (and some of our boards have *a lot of them*), as only the enumeration of pin names and the configuration array will need to be extended, without any need to write additional, repetitive code structures. Can you try to restructure the `Pins` package in this way?

There are also additional notes for each of our boards:

Arduino M0

The interrupt handlers are defined at the level of NVIC interrupts, not at the level of individual I/O lines and not even at the level of I/O ports. This means that a single procedure playing the role of the I/O interrupt handler has to use some logic in order to distinguish which of the I/O pins has met the condition (like rising or falling edge) to generate the interrupt.

Note also that I/O pins are mapped to EIC lines (EXTINTx), and this mapping can introduce conflicts when two I/O pins have the same associated EXTINTx line (see the *I/O Multiplexing and Considerations* chapter for the exact mapping chart) - that is, just reading the pending flags might not be enough to discover which pin has triggered the interrupt. The handler can therefore use some knowledge about the previous state of the pin to discover which pin exactly has changed its state. The next chapter explains how to manage such retention and sharing of state.

Arduino Due

The interrupt handlers are defined at the level of PIO interrupts, not at the level of individual I/O lines. This means that even though it is possible to have separate handlers for interrupts generated by pin 2 (PIOB) and pin 3 (PIOC), it is not possible with pins 3 and 4, as they belong to the same peripheral (PIOC). Normally this is not a problem - you can define a single procedure that handles events of similar nature (like buttons being pressed) and use some logic in the handler itself, like reading pins or checking interrupt status bit for each line, to figure out what was actually pressed. This grouping of interrupt sources is also relevant when interrupts sources are cleared, as the whole group is cleared with a single read of the Interrupt Status Register. This should not be a problem in practice.

STM32 Nucleo-32

For the purpose of interrupts, the I/O lines are multiplexed by EXTI controllers. You can see from the diagrams in the documentation that, for example, EXTI0 manages I/O lines PA0, PB0, PC0, and so on - which means that if one of them is selected, others will not take part in interrupt generation. This might be a problem with the Nucleo-32 board, where the possible conflict involves the following lines:

- PB0 with PF0 - these are Arduino connector pins 3 and 7,
- PB1 with PF1 - these are Arduino connector pins 6 and 8.

Make sure that these pin pairs are not used in a way that requires them to be independent interrupt sources.

STM32 Nucleo-144

For the purpose of interrupts and similarly to Nucleo-32, the I/O lines are multiplexed by EXTI controllers. You can see from the diagrams in the documentation that, for example, EXTI0 manages I/O lines PA0, PB0, PC0, and so on - which means that if one of them is selected, others will not take part in interrupt generation. This might be a problem with the Nucleo-144 board, where the possible conflict involves the following lines:

- PE13 with PF13 - these are Arduino connector pins 3 and 7,
- PF14 with PD14 - these are Arduino connector pins 4 and 10.

Make sure that these pin pairs are not used in a way that requires them to be independent interrupt sources.

Previous: Machine Code Insertions, next: Shared State.
See also Table of Contents.