8. Digital Input

We have already managed to configure two pins for digital output. This allows us to drive output state of selected pins, but is not enough for any reasonable control system that can react to external events. For this we also need the ability to read the state of selected pins.

Fortunately, most of the conceptual work was already done and the exploration paths established in previous chapters are still valid. Still, there is one thing that deserves additional explanation.

The general-purpose input pins can work in several modes that differ in their electrical properties. Note that not all modes are available in each microcontroller:

- Without any internal routing, which means that when the pin is left disconnected from any external voltage source, its state will be indeterminate - it might be occasionally seen as low or high depending on the electric fields around the device. This mode of operation should be used only when the pin is always connected to some deterministic signal source.
- With the internal pull-up (or pull-down) resistor, which ensures high (or low) state when the pin is not connected to anything - it is of course still possible to drive it with external voltage sources to both high and low states, but there is no indeterminate state when the pin is physically disconnected. This is useful as it simplifies electric connections for buttons and other simple passive sensors, which operate only in "open" and "closed" modes.

Neither of these modes is better or worse than the other, as they solve different problems and can be both useful in different designs - we will try to handle both of these two cases.

Note also that in all of our boards, the peripheral needs to be fed by the clock signal, otherwise reading the pin level will not be possible, and that clock is disabled by default after reset. This means that as part of the I/O configuration we also need to enable appropriate clock line - this was already necessary with the Nucleo boards even for simple output, but for digital input all microcontrollers require proper peripheral clocking.

In short, the above rules can be summarized as follows, differently for each board, but with some common patterns that should be easy to follow.

Arduino M0

Since the I/O pins are ready for operation by default, the configuration step does not need to involve enabling the clock signal, as is the case in other boards. The procedure for configuring the pin for digital input will be:

- instruct the microcontroller that we want the given pin to operate as input, by clearing the direction register by means of writing to GPIOx_DIRCLR,
- enable input and optionally pull-up resistor if appropriate - this is done via the GPIOx_PINCFGy register (bit 1 and 2, respectively), which is defined as a separate *byte* for each pin.

Note that the GPIOx_PINCFGy register should be accessed as an 8-bit entity - in order to make it easier we can add the following definition to the package `Registers`:

```
   type Byte is mod 2**8;
```

and we will use this type in later definitions of relevant register objects.

Once the pin is configured for input, and in order to read its current input state:

- read appropriate bit of the GPIOx_IN register, which contains the current pin state value.

Arduino Due

In order to configure the pin for digital input we need to:

- enable the clock for the appropriate peripheral (use the Arduino mapping documentation and chapter *Peripheral Identifiers* to find out which bits are relevant for your chosen pins),
- enable PIO control over the given pin - use the relevant PIOx_PER register for this,

- disable output control via PIOx_ODR - the output control is disabled by default, but let's do it for completeness and consistency with other settings,
- enable pull-up resistor if appropriate - this is done via the PIOx_PUER register (conversely, PIOx_PUDR disables it).

and in order to read the current input state:

- read appropriate bit of the PIOx_PDSR register, which contains the current pin state value.

STM32 Nucleo-32 and Nucleo-144

In order to configure the pin for digital input we need to:

- enable the clock for the selected I/O port (this is what we already did for output, check the documentation of the RCC_AHBENR or RCC_AHB1ENR registers to see which bits are responsible for each I/O port),
- write 00 to appropriate bits of the GPIOx_MODER register (this sets the input mode, needed only once),
- enable pull-up (or pull-down) resistor if needed with the GPIOx_PUPDR register - note that this register is operated in bit-pairs, similarly to GPIOx_MODER, and values that are of interest to use are 00 for no pull and 01 for pull-up.

and in order to read the current input state:

- read appropriate bit of the GPIOx_IDR register, which contains the current pin state value.

The Arduino-compatible boards have several rows of pins. We can focus on pins numbered from 2 to 13 - we have already discovered how to control pins 11 and 12, now we can extend this set.

Arduino M0

For Arduino M0 the controller mapping is:

- pin 2 maps to Port A, line 08,
- pin 3 maps to Port A, line 09,
- pin 4 maps to Port A, line 14,
- pin 5 maps to Port A, line 15,
- pin 6 maps to Port A, line 20,
- pin 7 maps to Port A, line 21,
- pin 8 maps to Port A, line 06,
- pin 9 maps to Port A, line 07,
- pin 10 maps to Port A, line 18,
- pin 11 maps to Port A, line 16 (we already have it),
- pin 12 maps to Port A, line 19 (we already have it),
- pin 13 maps to Port A, line 17.

Some of the registers that are needed here were already covered in previous examples, but our linker script will have to be extended to the following set of entries:

```
    GPIOA_DIRCLR = 0x41004404;
    GPIOA_DIRSET = 0x41004408;
    GPIOA_OUTCLR = 0x41004414;
    GPIOA_OUTSET = 0x41004418;
    GPIOA_IN = 0x41004420;
    GPIOA_PINCFG14 = 0x4100444e;
    GPIOA_PINCFG09 = 0x41004449;
    GPIOA_PINCFG08 = 0x41004448;
    GPIOA_PINCFG15 = 0x4100444f;
    GPIOA_PINCFG20 = 0x41004454;
    GPIOA_PINCFG21 = 0x41004455;
    GPIOA_PINCFG06 = 0x41004446;
    GPIOA_PINCFG07 = 0x41004447;
    GPIOA_PINCFG18 = 0x41004452;
    GPIOA_PINCFG16 = 0x41004450;
    GPIOA_PINCFG19 = 0x41004453;
    GPIOA_PINCFG17 = 0x41004451;
```

Fortunately, all pins that we want to cover are mapped to lines belonging to the same I/O port.

Arduino Due

For Arduino Due the controller mapping is:

- pin 2 maps to PIO B (peripheral ID 12, bit 12 of PMC_PCER0), line 25,
- pin 3 maps to PIO C (peripheral ID 13, bit 13 of PMC_PCER0), line 28,
- pin 4 maps to PIO C, line 26,
- pin 5 maps to PIO C, line 25,
- pin 6 maps to PIO C, line 24,
- pin 7 maps to PIO C, line 23,
- pin 8 maps to PIO C, line 22,
- pin 9 maps to PIO C, line 21,
- pin 10 maps to PIO C, line 29,
- pin 11 maps to PIO D (peripheral ID 14, bit 14 of PMC_PCER0), line 7 (we already have it),
- pin 12 maps to PIO D, line 8 (we already have it),
- pin 13 maps to PIO B, line 27.

Some of the registers that are needed here were already covered in previous examples, but our linker script will have to be extended by these entries:

```
PIOB_PER = 0x400E1000;
PIOB_OER = 0x400E1010;
PIOB_ODR = 0x400E1014;
PIOB_PUER = 0x400E1064;
PIOB_PUDR = 0x400E1060;
PIOB_SODR = 0x400E1030;
PIOB_CODR = 0x400E1034;
PIOB_PDSR = 0x400E103C;

PIOC_PER = 0x400E1200;
PIOC_OER = 0x400E1210;
PIOC_ODR = 0x400E1214;
PIOC_PUER = 0x400E1264;
PIOC_PUDR = 0x400E1260;
PIOC_SODR = 0x400E1230;
PIOC_CODR = 0x400E1234;
PIOC_PDSR = 0x400E123C;

PIOD_ODR = 0x400E1414;
PIOD_PUER = 0x400E1464;
PIOD_PUDR = 0x400E1460;
PIOD_PDSR = 0x400E143C;

PMC_PCER0 = 0x400E0610;
```

STM32 Nucleo-32

For Nucleo-32 the pin mapping is:

- pin 2 maps to port A, line PA12,
- pin 3 maps to port B, line PB0,
- pin 4 maps to port B, line PB7,
- pin 5 maps to port B, line PB6,
- pin 6 maps to port B, line PB1,
- pin 7 maps to port F, line PF0,
- pin 8 maps to port F, line PF1,
- pin 9 maps to port A, line PA8,
- pin 10 maps to port A, line PA11,
- pin 11 maps to port B, line PB5 (we already have it),
- pin 12 maps to port B, line PB4 (we already have it),
- pin 13 maps to port B, line PB3.

Some of the registers that are needed here were already covered in previous examples, but our linker script will have to be extended by these entries:

```
GPIOA_MODER = 0x48000000;
GPIOA_PUPDR = 0x4800000c;
GPIOA_IDR = 0x48000010;
GPIOA_BSRR = 0x48000018;
GPIOA_BRR = 0x48000028;
```

```
    GPIOB_MODER = 0x48000400;
    GPIOB_PUPDR = 0x4800040c;
    GPIOB_IDR = 0x48000410;
    GPIOB_BSRR = 0x48000418;
    GPIOB_BRR = 0x48000428;

    GPIOF_MODER = 0x48001400;
    GPIOF_PUPDR = 0x4800140c;
    GPIOF_IDR = 0x48001410;
    GPIOF_BSRR = 0x48001418;
    GPIOF_BRR = 0x48001428;
```

STM32 Nucleo-144

For Nucleo-144 the pin mapping is:

- pin 2 maps to port F, line PF15,
- pin 3 maps to port E, line PE13,
- pin 4 maps to port F, line PF14,
- pin 5 maps to port E, line PE11,
- pin 6 maps to port E, line PE9,
- pin 7 maps to port F, line PF13,
- pin 8 maps to port F, line PF12,
- pin 9 maps to port D, line PD15,
- pin 10 maps to port D, line PD14,
- pin 11 maps to port A, line PA7 (we already have it),
- pin 12 maps to port A, line PA6 (we already have it),
- pin 13 maps to port A, line PA5.

Some of the registers that are needed here were already covered in previous examples, but our linker script will have to be extended by these entries:

```
    GPIOA_MODER = 0x40020000;
    GPIOA_PUPDR = 0x4002000c;
    GPIOA_IDR = 0x40020010;
    GPIOA_BSRR = 0x40020018;

    GPIOD_MODER = 0x40020c00;
    GPIOD_PUPDR = 0x4002c0c;
    GPIOD_IDR = 0x40020c10;
    GPIOD_BSRR = 0x40020c18;

    GPIOE_MODER = 0x40021000;
    GPIOE_PUPDR = 0x4002100c;
    GPIOE_IDR = 0x40021010;
    GPIOE_BSRR = 0x40021018;

    GPIOF_MODER = 0x40021400;
    GPIOF_PUPDR = 0x4002140c;
    GPIOF_IDR = 0x40021410;
    GPIOF_BSRR = 0x40021418;
```

Apart from extending the linker scripts, the `Registers` package should be extended with appropriate definitions as well, appropriately for each board.

Adding the input capability requires an extension of the `Pins` package, which now contains the following procedure:

```
   procedure Enable_Output (Pin : in Pin_ID);
```

An obvious extension of this interface would be a new `Enable_Input` procedure, but with two different modes of operation we will use additional parameter to select the required mode (the two input modes seem to be common for our boards):

```
   type Input_Mode is (Direct, Pulled_Up);

   procedure Enable_Input (Pin : in Pin_ID; Mode : in Input_Mode);
```

Note that the choice of this interface is subjective and other options, like using separate procedures for each mode, could be used as well.

The `Enable_Output` procedure body (in the `pins.adb` file) will have to be extended to cover more pins, but the extension is straightforward. The new procedure for configuring input pins is more interesting.

Arduino M0

For Arduino M0 the `Enable_Input` can have the following structure:

```
procedure Enable_Input (Pin : in Pin_ID; Mode : in Input_Mode) is
begin
   case Pin is
      when Pin_2 =>
         Registers.GPIOA_DIRCLR := 2#100_0000_0000_0000#; -- bit 14

         if Mode = Pulled_Up then
            Registers.GPIOA_PINCFG16 := 2#0000_0110#;
            Registers.GPIOA_OUTSET := 2#100_0000_0000_0000#;
         else
            Registers.GPIOA_PINCFG16 := 2#0000_0010#;
         end if;

      --  other pins accordingly
      --  ...

   end case;
end Enable_Input;
```

Note that, interestingly, the output register GPIOA_OUTSET takes part in configuring the pull-up resistor as well.

Arduino Due

For Arduino Due the `Enable_Input` can have the following structure:

```
procedure Enable_Input (Pin : in Pin_ID; Mode : in Input_Mode) is
begin
   case Pin is
      when Pin_2 =>
         Registers.PMC_PCER0 :=
            2#1_0000_0000_0000#; -- bit 12
         Registers.PIOB_PER :=
            2#10_0000_0000_0000_0000_0000_0000#; -- bit 25
         Registers.PIOB_ODR :=
            2#10_0000_0000_0000_0000_0000_0000#; -- bit 25

         if Mode = Pulled_Up then
            Registers.PIOB_PUER :=
               2#10_0000_0000_0000_0000_0000_0000#; -- bit 25
         else
            Registers.PIOB_PUDR :=
               2#10_0000_0000_0000_0000_0000_0000#; -- bit 25
         end if;

      --  other pins accordingly
      --  ...

   end case;
end Enable_Input;
```

STM32 Nucleo-32 and Nucleo-144

For Nucleo-32 the `Enable_Input` can have the following structure:

```
procedure Enable_Input (Pin : in Pin_ID; Mode : in Input_Mode) is
   use type Registers.Word;
begin
   case Pin is
      when Pin_2 =>
         Registers.RCC_AHBENR := Registers.RCC_AHBENR
            or 2#10_0000_0000_0000_0000#;

         Registers.GPIOA_MODER := Registers.GPIOA_MODER
            and 2#1111_1100_1111_1111_1111_1111_1111_1111#;

         if Mode = Pulled_Up then
            Registers.GPIOA_PUPDR := (Registers.GPIOA_PUPDR
               and 2#1111_1100_1111_1111_1111_1111_1111_1111#)
                or 2#0000_0001_0000_0000_0000_0000_0000_0000#;
         else
            Registers.GPIOA_PUPDR := Registers.GPIOA_PUPDR
               and 2#1111_1100_1111_1111_1111_1111_1111_1111#;
         end if;

      --  other pins accordingly
      --  ...
   end case;
end Enable_Input;
```

The code structure for Nucleo-144 will be similar, differing in the use of RCC_AHB1ENR instead of RCC_AHBENR and in the actual bit patterns, appropriately to the pin mappings.

Last but not least, we need some operation for reading current state of the input pin. We can try with the following, complementary to the existing `Write` procedure.

Arduino M0

For Arduino Due, reading the state of input pin can be done as follows:

```
   procedure Read (Pin : in Pin_ID; State : out Boolean) is
      Data : Registers.Word;
   begin
      case Pin is
         when Pin_2 =>
            Data := Registers.GPIOA_IN;
            State := (Data and 2#100_0000_0000_0000#) /= 0;

         --  other pins accordingly
         --  ...

      end case;
   end Read;
```

Arduino Due

For Arduino Due, reading the state of input pin can be done as follows:

```
   procedure Read (Pin : in Pid_ID; State : out Boolean) is
      Data : Registers.Word;
   begin
      case Pin is
         when Pin_2 =>
            Data := Registers.PIOB_PDSR;
            State :=
               (Data and 2#10_0000_0000_0000_0000_0000_0000#) /= 0;

         --  other pins accordingly
         --  ...

      end case;
   end Read;
```

STM32 Nucleo-32 and Nucleo-144

For Nucleo-32, reading the state of input pin can be done as follows:

```
   procedure Read (Pin : in Pid_ID; State : out Boolean) is
      Data : Registers.Word;
   begin
      case Pin is
         when Pin_2 =>
            Data := Registers.GPIOB_IDR;
            State :=
               (Data and 2#1_0000_0000_0000#) /= 0;

         --  other pins accordingly
         --  ...

      end case;
   end Read;
```

As already noticed above, the code for Nucleo-144 will be very similar and will differ only in actual ports and bit patterns.

Note that in the `Read` procedure the `State` parameter has an output mode, similarly to our earlier procedure for reading random numbers.

The new procedures and register definitions allow us to configure any of the pins numbered from 2 to 13 for both input and output as well as to read and write them. Just for the sake of basic testing, the following program copies the logical state from pin 2 (configured as input) to pin 3 (configured as output):

```
with Pins;

package body Program is

   procedure Run is
      State : Boolean;
   begin
      Pins.Enable_Input (Pins.Pin_2, Pins.Pulled_Up);
      Pins.Enable_Output (Pins.Pin_3);

      loop
         Pins.Read (Pins.Pin_2, State);
         Pins.Write (Pins.Pin_3, State);
      end loop;
   end Run;

end Program;
```

There were no new Ada elements in this chapter, but with the complete control over the whole row of digital pins we are now well prepared for some more realistic projects involving interactions with the external world.

Previous: Random Numbers, next: Finite State Machines, Part 1.
See also Table of Contents.

Did you find this article interesting? Share it!

SHARE