5. Digital Output

In this chapter we will attempt to actually do something observable with the board - we will control the state of digital output pins.

Both Atmel and STM32 microcontrollers are very flexible and offer lots of functionality - in fact, they have more internal functions than can be exposed on their pins at the same time. For this reason most of the pins are multi-purpose, relying on configurable multiplexing to assign the given physical pin to the given microcontroller function. This also means that setting the state of any given output pin requires a bit of preparation and the microcontroller has to be configured appropriately so that the given pins can be used for digital output and not, for example, for network communication. Further, there are several ways in which digital I/O lines can operate and these have to be selected as well.

For the sake of example, let's focus on two pins on the standard Arduino connector: 11 and 12. These pins are present on each of our boards.

We need to figure out the pin mapping for the given board - that is, how microcontroller pins are traced to the board external connectors. Such mappings can be derived from the schematics published on the Arduino website or from the diagrams delivered as part of the Nucleo packages.

Arduino M0

We can see from the schematics that board pins 11 and 12 are traced to microcontroller pins 25 (named as PA16) and 28 (PA19), respectively. These are general-purpose I/O lines, known simply as GPIO. Both of these lines belong to so-called Port A, which is a parallel port that can control multiple I/O lines. The same physical lines can be used for other purposes as well, so we will have to configure the chip to use them only for digital I/O.

The following steps have to take place in order to control the output on the selected pins:

- We can rely on the default microcontroller settings (these become active after each system startup), where I/O pins are controlled by GPIO and alternative peripherals are not active. This observation simplifies the configuration of the device. More information on this can be found in the documentation in sections devoted to function multiplexing.
- We have to instruct the microcontroller that we want to drive the output state of the pin. The relevant registers that have to be used is called DIRSET and each pin has its associated bit in that register.
- Once the pin is configured properly, we can set its desired output state with the help of registers OUTSET (for setting the output to the high state) and OUTCLR (for setting the output to the low state). Again, each pin has its own bit in these registers.

The above register names are meaningful only in relation to the whole group of registers assigned for controlling the given port, so a little bit of documentation digging is needed to figure out the actual register addresses that we need to use. From the product mapping and peripherals configuration data we can see that registers devoted to controlling ports occupy memory space from 0x41004400 and from the register summary table for I/O ports we can see that Port A has offset 0x0 from that range. This allows us to disambiguate the register names and compute their actual addresses:

- GPIOA_DIRSET (pin direction for Port A) has address 0x41004408,
- GPIOA_OUTSET (set high) has address 0x41004418,
- GPIOA_OUTCLR (set low or clear) has address 0x41004414.

As these registers have separate bits for each I/O line, we should focus on bits 16 and 19 in order to control the microcontroller pins 25 and 28, which are directly connected to the Arduino connector pins 11 and 12.

To summarize, in order to configure Arduino pin 11 for output and set its state to "high" we need to:

- write 1 to bit 16 of GPIOA_OUTDIR (this needs to be done once, perhaps at initialization time),
- write 1 to bit 16 of GPIOA_OUTSET (this can be repeated multiple times).

Similarly, in order to configure Arduino pin 12 for output and set its state to "low" we need to:

- write 1 to bit 19 of GPIOA_OUTDIR (once),
- write 1 to bit 19 of GPIOA_OUTCLR (perhaps multiple times).

Of course, bits in GPIOA_OUTSET and GPIOA_OUTCLR can be set any time whenever there is a need to change the output state to "high" or "low".

Arduino Due

From the Arduino board pin mapping documentation page or from the schematics published on the Arduino web site, we can see that pins 11 and 12 on the board correspond to microcontroller's pins PD7 and PD8. These are general-purpose I/O lines (GPIO) and are managed by the PIO controller. There are several PIO controllers, but as you can see from section about multiplexing in the microcontroller documentation, pins PD7 and PD8 belong to controller D, also known as PIOD, and can be shared with some other peripheral functionality. The PIOD controller manages 10 I/O lines in total in the microcontroller variant that is used in Arduino Due (it is the 144-pin package) and the rules for configuration and operation are similar for all of these lines.

The following actions have to be taken in order to control the output state of our chosen pins:

- We have to instruct the microcontroller that we do not want alternative peripheral functions and we want the PIO to take over - according to section about I/O peripheral function selection in the Atmel documentation, the register responsible for this is called PIO_PER (PIO Enable Register), and in the case of controller D the full name of the relevant register is PIOD_PER.
- We have to instruct the microcontroller that we want to actually drive the output state of the line (otherwise the line could be only used as input) and from section about output control we know that the relevant configuration register is PIO_OER (Output Enable Register), or PIOD_OER for controller D.
- Once the two steps above are done, we can set the desired output state and from the same section as above we can see that registers responsible for it are PIOD_SODR (Set Output Data Register) and PIOD_CODR (Clear Output Data Register).

All of the registers described above have separate bits for each controlled line. These are bits number 7 and 8 in our case.

To summarize, in order to configure the Arduino Due pin 11 for output and set its state to "high" we need to:

- write 1 to bit 7 of PIOD_PER (this needs to be done once at initialization time),
- write 1 to bit 7 of PIOD_OER (this also needs to be done once),
- write 1 to bit 7 of PIOD_SODR (this can be repeated any time).

Similarly, to configure Arduino Due pin 12 for output and set its state to "low" we need to:

- write 1 to bit 8 of PIOD_PER,
- write 1 to bit 8 of PIOD_OER,
- write 1 to bit 8 of PIOD_CODR

and of course we can write 1 to appropriate bits in PIO_SODR (set) and PIO_CODR (clear) whenever we want to change the output state.

The only missing piece of information is the location of each register and from sections about product mapping and the register summary we know that:

- PIOD_PER is located at 0x400E1400,
- PIOD_OER is located at 0x400E1410,
- PIOD_SODR is located at 0x400E1430 and
- PIOD_CODR is located at 0x400E1434.

STM32 Nucleo-32

The pin mapping documentation for Nucleo boards is part of the package and on the diagram we can see that pins 11 and 12 on the board correspond to microcontroller's pins PB5 and PB4. These are called general-purpose I/O lines (GPIO). There are several GPIO *ports* that group multiple lines and pins PB5 and PB4 belong to port B, and can be shared with some other peripheral functionality, but such sharing is disabled by default after reset - we will benefit from this fact to simplify our own code.

The following actions have to be taken in order to control the output state of our chosen pins:

- We have to instruct the microcontroller that we want to use the GPIO peripheral, and that the clock signal has to be delivered to it. Without proper clocking the peripheral would be non-functional. The register that is responsible for controlling clocks for I/O ports is called RCC_AHBENR and each I/O port has a dedicated bit in that register - in the case of port B, it is bit 18.
- We have to instruct the microcontroller that we want to actually drive the output state of the line (otherwise the line could be only used as input) and from GPIO functional description we know that the relevant configuration register is GPIOx_MODER (Port Mode Register), or GPIOB_MODER for port B.

- Once the two steps above are done, we can set the desired output state and from the same section as above we can see that registers responsible for it are GPIOB_BSRR (Port Bit Set/Reset Register), which has separate bits dedicated to setting the output state to "high" (these are bits in the lower half of the register) or "low" (these are in the upper half). Alternatively, we can use separate GPIOB_BRR register to reset the bits, with bit numbering consistent with GPIOB_BSRR.

All of the registers described above have separate bits for each controlled line, but they are not consistently numbered. The GPIOx_MODER register uses *pairs* of bits for each I/O line in the given port, whereas other registers tend to use single bit mappings for each line.

To be exact, in order to configure the Nucleo-32 pin 11 for output and set its state to "high" we need to:

- write 1 to bit 18 of the RCC_AHBENR (this needs to be done once at initialization time),
- write 01 into the bit pair 10-11 of GPIOB_MODER, leaving other bits unchanged (this needs to be done once),
- write 1 to bit 5 of GPIOB_BSRR (this can be repeated any time).

Similarly, to configure pin 12 for output and set its state to "low" we need to:

- write 1 to bit 18 of the RCC_AHBENR (this needs to be done once at initialization time),
- write 01 into the bit pair 8-9 of GPIOB_MODER, leaving other bits unchanged (this needs to be done once),
- write 1 to bit 4 of GPIOB_BRR (this can be repeated any time).

and of course we can write 1 to appropriate bits in GPIOB_BSRR (set) and GPIOB_BRR (reset) whenever we want to change the output state.

The only missing piece of information is the location of each register and from sections about memory mapping and the GPIO register map we know that:

- RCC_AHBENR is located at 0x40021014,
- GPIOB_MODER is located at 0x48000400,
- GPIOB_BSRR is located at 0x48000418,
- GPIOB_BRR is located at 0x48000428.

STM32 Nucleo-144

The pin mapping documentation for Nucleo boards is part of the package and on the diagram we can see that pins 11 and 12 on the board correspond to microcontroller's pins PA7 and PA6. These are called general-purpose I/O lines (GPIO). There are several GPIO *ports* that group multiple lines and pins PA7 and PA6 belong to port A, and can be shared with some other peripheral functionality, but such sharing is disabled by default after reset - we will benefit from this fact to simplify our own code.

The following actions have to be taken in order to control the output state of our chosen pins:

- We have to instruct the microcontroller that we want to use the GPIO peripheral, and that the clock signal has to be delivered to it. Without proper clocking the peripheral would be non-functional. The register that is responsible for controlling clocks for I/O ports is called RCC_AHB1ENR and each I/O port has a dedicated bit in that register - in the case of port A, it is bit 0.
- We have to instruct the microcontroller that we want to actually drive the output state of the line (otherwise the line could be only used as input) and from GPIO functional description we know that the relevant configuration register is GPIOx_MODER (Port Mode Register), or GPIOA_MODER for port A.
- Once the two steps above are done, we can set the desired output state and from the same section as above we can see that a single register responsible for it is GPIOA_BSRR (Port Bit Set/Reset Register), which has separate bits dedicated to setting the output state to "high" (these are bits in the lower half of the register) or "low" (these are in the upper half).

All of the registers described above have separate bits for each controlled line, but they are not consistently numbered. The GPIOx_MODER register uses *pairs* if bits for each I/O line in the given port, whereas other registers tend to use single bit mappings for each line.

To be exact, in order to configure the Nucleo-144 pin 11 for output and set its state to "high" we need to:

- write 1 to bit 0 of the RCC_AHB1ENR (this needs to be done once at initialization time),
- write 01 into the bit pair 14-15 of GPIOA_MODER, leaving other bits unchanged (this needs to be done once),
- write 1 to bit 7 of GPIOA_BSRR (this can be repeated any time).

Similarly, to configure pin 12 for output and set its state to "low" we need to:

- write 1 to bit 0 of the RCC_AHB1ENR (this needs to be done once at initialization time),

- write 01 into the bit pair 12-13 of GPIOA_MODER, leaving other bits unchanged (this needs to be done once),
- write 1 to bit 22 (that is, 6 + 16) of GPIOA_BSRR (this can be repeated any time)

and of course we can write 1 to appropriate bits in GPIOA_BSRR (set and reset) whenever we want to change the output state.

The only missing piece of information is the location of each register and from sections about memory mapping and the GPIO register map we know that:

- RCC_AHB1ENR is located at 0x40023814,
- GPIOA_MODER is located at 0x40020000,
- GPIOA_BSRR is located at 0x40020018.

There are several alternative ways in which controlling of individual registers can be achieved in Ada.

One way is to define a pointer value (this is called access value in Ada) for the appropriate memory location and dereference it for writing to the given register. This approach is mostly popular among C and C++ programmers, and an example code in C could be (here shown with registers from Arduino M0):

```
    volatile uint32_t * const GPIOA_DIRSET = (uint32_t *)0x41004408;
    *GPIOA_DIRSET = 0x1 << 16u;
```

This is also possible in Ada, but is not considered to be a good programming practice and we will not explore it.

Another option is to declare a regular variable that will mirror the physical register and use so-called representation aspects to instruct the compiler where, physically, this variable is supposed to be located. This could look similar to this (for Arduino M0):

```
   GPIOA_DIRSET : Word;
   pragma Volatile (GPIOA_DIRSET);
   for GPIOA_DIRSET'Address use 16#41004408#;

   -- and then:
   GPIOA_DIRSET := 2#1_0000_0000_0000_0000#;
```

Such construct is quite popular in Ada (note that it does not rely on pointers and entity GPIOA_DIRSET, being a variable, better reflects the fact that a register is an actual object), but has a drawback of exposing physical addresses in source code.

We will use a third approach, which allows separation of concerns: the source code will be responsible for business logic and the linker script will take responsibility for memory layout and detailed object placement. The advantage of this approach is that handling of addresses and memory locations is kept in a single place and the program source code is not polluted with hardware details.

That third approach will rely on the linker to manage addresses, so we need to extend the linker script with some new elements. This of course will be different for each of our boards.

Arduino M0

For Arduino M0 the full linker script will look like this:

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)

SECTIONS
{
    GPIOA_DIRSET = 0x41004408;
    GPIOA_OUTSET = 0x41004418;
    GPIOA_OUTCLR = 0x41004414;

    .vectors 0x00004000 :
    {
        LONG(0x20008000)
        LONG(run + 1)
        FILL(0)
    }

    .text 0x00004100 :
    {
        *(.text)
    }
}
```

Arduino Due

For Arduino Due the linker script will look like this:

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)

SECTIONS
{
    PIOD_PER = 0x400E1400;
    PIOD_OER = 0x400E1410;
    PIOD_SODR = 0x400E1430;
    PIOD_CODR = 0x400E1434;

    .vectors 0x00080000 :
    {
        LONG(0x20088000)
        LONG(run + 1)
        FILL(0)
    }

    .text 0x00080100 :
    {
        *(.text)
    }
}
```

STM32 Nucleo-32

For Nucleo-32 the linker script will look like this:

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)

SECTIONS
{
    RCC_AHBENR = 0x40021014;
    GPIOB_MODER = 0x48000400;
    GPIOB_BSRR = 0x48000418;
    GPIOB_BRR = 0x48000428;

    .vectors 0x00002000 :
    {
        LONG(0x20008000)
        LONG(run + 1)
        FILL(0)
    }

    .text 0x00002100 :
    {
        *(.text)
    }
}
```

STM32 Nucleo-144

For Nucleo-144 the linker script will look like this:

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)

SECTIONS
{
    RCC_AHB1ENR = 0x40023830;
    GPIOA_MODER = 0x40020000;
    GPIOA_BSRR = 0x40020018;

    .vectors 0x08000000 :
    {
        LONG(0x20010000)
        LONG(run + 1)
        FILL(0)
    }

    .text 0x08000200 :
    {
        *(.text)
    }
}
```

Note the sequence of address assignments at the beginning of the SECTIONS block. They provide the necessary information for

the linker so that whenever symbols like `GPIOB_MODER` or `PIOD_PER` appear in the linked object files, the linker will resolve them to proper addresses. You can add as many assignments like this as you need.

Then, we need to declare Ada variables that will represent these registers, but before that we have to decide on the appropriate type for these variables (and consequently for all operations that will be done on them).

The type that represents 32-bit unsigned value can be defined in Ada like this:

```
   type Word is mod 2**32;
```

This is called a modular type and has valid values in the range from 0 to 2^32 - 1 with wrap semantics for under- and overflows (in essence this is equivalent to `uint32_t` type from C and C++). Having such a type we can make appropriate variable declarations like (for Arduino M0):

```
   GPIOA_DIRSET : Word;
   pragma Volatile (GPIOA_DIRSET);
   pragma Import (C, GPIOA_DIRSET, "GPIOA_DIRSET");
```

The above declares a variable named `GPIOA_DIRSET` of type `Word` and instructs the compiler that all accesses to this variable are meaningful (so that the compiler should not eliminate writes or reads to this variable as it could otherwise do in the process of code optimization) and, most importantly, that this variable already has allocated space and is known externally under the name `GPIOA_DIRSET`. You can compare the `pragma Import` here with `pragma Export` that was used for the `Run` procedure - in the case of procedure `Run` we wanted to provide the name to external tools like linker and in the case of `GPIOA_DIRSET` we want to rely on the linker to manage memory placement for this variable.

We can try to put all these pieces together in our program (file `program.adb`).

Arduino M0

```
package body Program is

   type Word is mod 2**32;

   GPIOA_DIRSET : Word;
   pragma Volatile (GPIOA_DIRSET);
   pragma Import (C, GPIOA_DIRSET, "GPIOA_DIRSET");

   GPIOA_OUTSET : Word;
   pragma Volatile (GPIOA_OUTSET);
   pragma Import (C, GPIOA_OUTSET, "GPIOA_OUTSET");

   GPIOA_OUTCLR : Word;
   pragma Volatile (GPIOA_OUTCLR);
   pragma Import (C, GPIOA_OUTCLR, "GPIOA_OUTCLR");

   procedure Run is
   begin
      -- configure and set high state for PA16 (for pin 11):
      GPIOA_DIRSET := 2#1_0000_1000_0000_0000#;
      GPIOA_OUTSET := 2#1_0000_1000_0000_0000#;

      -- configure and set low state for PA19 (for pin 12):
      GPIOA_DIRSET := 2#1000_0000_0000_0000_0000#;
      GPIOA_OUTCLR := 2#1000_0000_0000_0000_0000#;

      loop
         null;
      end loop;
   end Run;

end Program;
```

Arduino Due

Our basic I/O program for Arduino Due is:

```
package body Program is

   type Word is mod 2**32;

   PIOD_PER : Word;
   pragma Volatile (PIOD_PER);
   pragma Import (C, PIOD_PER, "PIOD_PER");

   PIOD_OER : Word;
   pragma Volatile (PIOD_OER);
```

```
   pragma Import (C, PIOD_OER, "PIOD_OER");

   PIOD_SODR : Word;
   pragma Volatile (PIOD_SODR);
   pragma Import (C, PIOD_SODR, "PIOD_SODR");

   PIOD_CODR : Word;
   pragma Volatile (PIOD_CODR);
   pragma Import (C, PIOD_CODR, "PIOD_CODR");

   procedure Run is
   begin
      -- configure and set high state for PD7 (for pin 11):
      PIOD_PER := 2#1000_0000#;
      PIOD_OER := 2#1000_0000#;
      PIOD_CODR := 2#1000_0000#;

      -- configure and set low state for PD8 (for pin 12):
      PIOD_PER := 2#1_0000_0000#;
      PIOD_OER := 2#1_0000_0000#;
      PIOD_CODR := 2#1_0000_0000#;

      loop
         null;
      end loop;
   end Run;

end Program;
```

STM32 Nucleo-32

Our basic I/O program for Nucleo-32 is:

```
package body Program is

   type Word is mod 2**32;

   RCC_AHBENR : Word;
   pragma Volatile (RCC_AHBENR);
   pragma Import (C, RCC_AHBENR, "RCC_AHBENR");

   GPIOB_MODER : Word;
   pragma Volatile (GPIOB_MODER);
   pragma Import (C, GPIOB_MODER, "GPIOB_MODER");

   GPIOB_BSRR : Word;
   pragma Volatile (GPIOB_BSRR);
   pragma Import (C, GPIOB_BSRR, "GPIOB_BSRR");

   GPIOB_BRR : Word;
   pragma Volatile (GPIOB_BRR);
   pragma Import (C, GPIOB_BRR, "GPIOB_BRR");

   procedure Run is
   begin
      -- configure clock for I/O port B:
      RCC_AHBENR := RCC_AHBENR or 2#100_0000_0000_0000_0000#;

      -- configure and set high state for PB5 (for pin 11):
      GPIOB_MODER := (GPIOB_MODER
         and 2#1111_1111_1111_1111_1111_0011_1111_1111#)
          or 2#0000_0000_0000_0000_0000_0100_0000_0000#;
      GPIOB_BSRR := 2#10_0000#;

      -- configure and set low state for PB4 (for pin 12):
      GPIOB_MODER := (GPIOB_MODER
         and 2#1111_1111_1111_1111_1111_1100_1111_1111#)
          or 2#0000_0000_0000_0000_0000_0001_0000_0000#;
      GPIOB_BRR := 2#1_0000#;

      loop
         null;
      end loop;
   end Run;

end Program;
```

STM32 Nucleo-144

Our basic I/O program for Nucleo-144 is:

```
package body Program is

   type Word is mod 2**32;
```

```
   RCC_AHB1ENR : Word;
   pragma Volatile (RCC_AHB1ENR);
   pragma Import (C, RCC_AHB1ENR, "RCC_AHB1ENR");

   GPIOA_MODER : Word;
   pragma Volatile (GPIOA_MODER);
   pragma Import (C, GPIOA_MODER, "GPIOA_MODER");

   GPIOA_BSRR : Word;
   pragma Volatile (GPIOA_BSRR);
   pragma Import (C, GPIOA_BSRR, "GPIOA_BSRR");

   procedure Run is
   begin
      -- configure clock for I/O port A:
      RCC_AHB1ENR := RCC_AHB1ENR or 2#1#;

      -- configure and set high state for PA7 (for pin 11):
      GPIOA_MODER := (GPIOA_MODER
         and 2#1111_1111_1111_1111_0011_1111_1111_1111#)
          or 2#0000_0000_0000_0000_0100_0000_0000_0000#;
      GPIOA_BSRR := 2#1000_0000#;

      -- configure and set low state for PA6 (for pin 12):
      GPIOA_MODER := (GPIOB_MODER
         and 2#1111_1111_1111_1111_1100_1111_1111_1111#)
          or 2#0000_0000_0000_0000_0001_0000_0000_0000#;
      GPIOA_BSRR := 2#100_0000_0000_0000_0000_0000#;

      loop
         null;
      end loop;
   end Run;

end Program;
```

The programs above, together with the `program.ads` specification file that we have written previously, is complete and does what it was supposed to do: it configures pin 11 and 12 for output and sets high and low states on them.

There are several new Ada elements here that should be explained:

- variables representing registers and their type were defined at the package scope, outside of the procedure `Run`, which can access them,
- the lines beginning with "`--`" are comments,
- "`2#1000_0000#`" is a binary literal corresponding to decimal value 128 (bit 7 is set), whereas "`2#1_0000_0000#`" is a binary literal corresponding to decimal value 256 (bit 8 is set), underscores are optional and are used only for readability.

This program is complete and since it uses the same number of files as before (`program.ads`, `program.adb` and `flash.ld`), you can try to compile, link and upload it using the same steps as were already described. We should, however, take the opportunity to refactor this code in anticipation of further growth and in order to abstract a bit the hardware differences between our boards - after all, it is useful to think about pin 11 on the Arduino connector as pin 11, without needing to work at the very detailed level of microcontroller registers, which, as you can see, are different and have different usage patterns on each microcontroller.

First, we will create a separate package for utility procedures and we will put the final spinning loop there. The package specification (`utils.ads`) is:

```
package Utils is

   procedure Spin_Indefinitely;

end Utils;
```

and the package body (`utils.adb`) is:

```
package body Utils is

   procedure Spin_Indefinitely is
   begin
      loop
         null;
      end loop;
   end Spin_Indefinitely;

end Utils;
```

Note that we do not need any `pragma Export` for the `Spin_Indefinitely` procedure, because it will not be exposed to the

tools.

We can also group register definitions in a separate package called `Registers`, in the `registers.ads` file, separately for each board:

Arduino M0

For Arduino M0, our `Registers` package will have the following definitions:

```
package Registers is

   type Word is mod 2**32;

   GPIOA_DIRSET : Word;
   pragma Volatile (GPIOA_DIRSET);
   pragma Import (C, GPIOA_DIRSET, "GPIOA_DIRSET");

   GPIOA_OUTSET : Word;
   pragma Volatile (GPIOA_OUTSET);
   pragma Import (C, GPIOA_OUTSET, "GPIOA_OUTSET");

   GPIOA_OUTCLR : Word;
   pragma Volatile (GPIOA_OUTCLR);
   pragma Import (C, GPIOA_OUTCLR, "GPIOA_OUTCLR");

end Registers;
```

Arduino Due

For Arduino Due, our `Registers` package will have the following definitions:

```
package Registers is

   type Word is mod 2**32;

   PIOD_PER : Word;
   pragma Volatile (PIOD_PER);
   pragma Import (C, PIOD_PER, "PIOD_PER");

   PIOD_OER : Word;
   pragma Volatile (PIOD_OER);
   pragma Import (C, PIOD_OER, "PIOD_OER");

   PIOD_SODR : Word;
   pragma Volatile (PIOD_SODR);
   pragma Import (C, PIOD_SODR, "PIOD_SODR");

   PIOD_CODR : Word;
   pragma Volatile (PIOD_CODR);
   pragma Import (C, PIOD_CODR, "PIOD_CODR");

end Registers;
```

STM32 Nucleo-32

For Nucleo-32, our `Registers` package will have the following definitions:

```
package Registers is

   type Word is mod 2**32;

   RCC_AHBENR : Word;
   pragma Volatile (RCC_AHBENR);
   pragma Import (C, RCC_AHBENR, "RCC_AHBENR");

   GPIOB_MODER : Word;
   pragma Volatile (GPIOB_MODER);
   pragma Import (C, GPIOB_MODER, "GPIOB_MODER");

   GPIOB_BSRR : Word;
   pragma Volatile (GPIOB_BSRR);
   pragma Import (C, GPIOB_BSRR, "GPIOB_BSRR");

   GPIOB_BRR : Word;
   pragma Volatile (GPIOB_BRR);
   pragma Import (C, GPIOB_BRR, "GPIOB_BRR");

end Registers;
```

STM32 Nucleo-144

For Nucleo-32, our `Registers` package will have the following definitions:

```
package Registers is

   type Word is mod 2**32;

   RCC_AHB1ENR : Word;
   pragma Volatile (RCC_AHB1ENR);
   pragma Import (C, RCC_AHB1ENR, "RCC_AHB1ENR");

   GPIOA_MODER : Word;
   pragma Volatile (GPIOA_MODER);
   pragma Import (C, GPIOA_MODER, "GPIOA_MODER");

   GPIOA_BSRR : Word;
   pragma Volatile (GPIOA_BSRR);
   pragma Import (C, GPIOA_BSRR, "GPIOA_BSRR");

end Registers;
```

Note that there is no need for `registers.adb` file, as there is no need to provide any more information to this package, the definitions in the specification file are already complete.

We can also define a separate package for managing pins - with no surprise we can call it `Pins` and write the following `pins.ads` specification file:

```
package Pins is

   type Pin_ID is ( Pin_11, Pin_12 );

   procedure Enable_Output (Pin : in Pin_ID);

   procedure Write (Pin : in Pin_ID; State : in Boolean);

end Pins;
```

In this package specification we have defined another type for identifying Arduino pins - the `Pin_ID` type is an enumeration type with two values. An alternative approach would be to use some numeric type for naming pins, but this would be less readable. Note how procedures `Enable_Output` and `Write` rely on this type in their parameters. We can reuse the existing `Boolean` type for representing output states (truth value for "high" and false value for "low").

The body of this package (in file `pins.adb`) will refer to hardware registers and will be different for each of our boards.

Arduino M0

For Arduino Due the `pins.adb` file can look like:

```
with Registers;

package body Pins is

   procedure Enable_Output (Pin : in Pin_ID) is
   begin
      case Pin is
         when Pin_11 =>
            Registers.GPIOA_DIRSET := 2#1_0000_1000_0000_0000#;

         when Pin_12 =>
            Registers.GPIOA_DIRSET := 2#1000_0000_0000_0000_0000#;
      end case;
   end Enable_Output;

   procedure Write (Pin : in Pin_ID; State : Boolean) is
   begin
      case Pin is
         when Pin_11 =>
            if State then
               Registers.GPIOA_OUTSET := 2#1_0000_1000_0000_0000#;
            else
               Registers.GPIOA_OUTCLR := 2#1_0000_1000_0000_0000#;
            end if;

         when Pin_12 =>
            if State then
               Registers.GPIOA_OUTSET := 2#1000_0000_0000_0000_0000#;
            else
               Registers.GPIOA_OUTCLR := 2#1000_0000_0000_0000_0000#;
            end if;
      end case;
   end Write;
```

```
end Pins;
```

Arduino Due

For Arduino Due the `pins.adb` file can look like:

```
with Registers;

package body Pins is

   procedure Enable_Output (Pin : in Pin_ID) is
   begin
      case Pin is
         when Pin_11 =>
            Registers.PIOD_PER := 2#1000_0000#;
            Registers.PIOD_OER := 2#1000_0000#;

         when Pin_12 =>
            Registers.PIOD_PER := 2#1_0000_0000#;
            Registers.PIOD_OER := 2#1_0000_0000#;
      end case;
   end Enable_Output;

   procedure Write (Pin : in Pin_ID; State : Boolean) is
   begin
      case Pin is
         when Pin_11 =>
            if State then
               Registers.PIOD_SODR := 2#1000_0000#;
            else
               Registers.PIOD_CODR := 2#1000_0000#;
            end if;

         when Pin_12 =>
            if State then
               Registers.PIOD_SODR := 2#1_0000_0000#;
            else
               Registers.PIOD_CODR := 2#1_0000_0000#;
            end if;
      end case;
   end Write;

end Pins;
```

STM32 Nucleo-32

For Nucleo-32 the `pins.adb` file can look like:

```
with Registers;

package body Pins is

   procedure Enable_Output (Pin : in Pin_ID) is
      use type Registers.Word;
   begin
      RCC_AHBENR := RCC_AHBENR or 2#100_0000_0000_0000_0000#;

      case Pin is
         when Pin_11 =>
            GPIOB_MODER := (GPIOB_MODER
               and 2#1111_1111_1111_1111_1111_0011_1111_1111#)
                or 2#0000_0000_0000_0000_0000_0100_0000_0000#;

         when Pin_12 =>
            GPIOB_MODER := (GPIOB_MODER
               and 2#1111_1111_1111_1111_1111_1100_1111_1111#)
                or 2#0000_0000_0000_0000_0000_0001_0000_0000#;
      end case;
   end Enable_Output;

   procedure Write (Pin : in Pin_ID; State : Boolean) is
   begin
      case Pin is
         when Pin_11 =>
            if State then
               GPIOB_BSRR := 2#10_0000#;
            else
               GPIOB_BRR := 2#10_0000#;
            end if;

         when Pin_12 =>
            if State then
               GPIOB_BSRR := 2#1_0000#;
```

```
            else
               GPIOB_BRR := 2#1_0000#;
            end if;
      end case;
   end Write;

end Pins;
```

STM32 Nucleo-144

For Nucleo-144 the `pins.adb` file can look like:

```
with Registers;

package body Pins is

   procedure Enable_Output (Pin : in Pin_ID) is
      use type Registers.Word;
   begin
      Registers.RCC_AHB1ENR := Registers.RCC_AHB1ENR or 2#1#;

      case Pin is
         when Pin_11 =>
            Registers.GPIOA_MODER := (Registers.GPIOA_MODER
               and 2#1111_1111_1111_1111_0011_1111_1111_1111#)
                or 2#0000_0000_0000_0000_0100_0000_0000_0000#;

         when Pin_12 =>
            Registers.GPIOA_MODER := (Registers.GPIOA_MODER
               and 2#1111_1111_1111_1111_1100_1111_1111_1111#)
                or 2#0000_0000_0000_0000_0001_0000_0000_0000#;
      end case;
   end Enable_Output;

   procedure Write (Pin : in Pin_ID; State : Boolean) is
   begin
      case Pin is
         when Pin_11 =>
            if State then
               Registers.GPIOA_BSRR := 2#1000_0000#;
            else
               Registers.GPIOA_BSRR :=
                  2#1000_0000_0000_0000_0000_0000#;
            end if;

         when Pin_12 =>
            if State then
               Registers.GPIOA_BSRR := 2#100_0000#;
            else
               Registers.GPIOA_BSRR :=
                  2#100_0000_0000_0000_0000_0000#;
            end if;
      end case;
   end Write;

end Pins;
```

Note that the `Pins` package refers to definitions from the `Registers` package - first it needs to announce such dependency with the `with Registers;` clause and then it needs to qualify relevant names with the package name, like in `Registers.PIOD_PER` - moreover, to access operations of the type defined in another package, the user package needs to introduce those operations by means of `use type` directive, which you can see at the beginning of `Enable_Output`. It is possible to write code without such full qualifications (with some additional arrangements), but qualifications are considered to be a good programming practice and we will keep them.

The `case`/`when` construct is similar to `switch` statements from other languages and allows handling one of many possible values of a given expression. Similarly to the `switch` statement, it is a perfect tool for dealing with enumeration values.

Now our main program (in `program.adb`) can have a more organized structure:

```
with Pins;
with Utils;

package body Program is

   procedure Run is
   begin
      Pins.Enable_Output (Pins.Pin_11);
      Pins.Enable_Output (Pins.Pin_12);

      Pins.Write (Pins.Pin_11, True);
      Pins.Write (Pins.Pin_12, False);
```

```
      Utils.Spin_Indefinitely;
   end Run;

end Program;
```

This program is *portable* in the sense that it has the same form independently on the target board - the necessary Hardware Abstraction Layer (HAL) is provided by other files.

Note that our main package announces dependency on packages `Pins` and `Utils` with appropriate `with` clauses at the beginning.

Our program is composed of several separate files, so we will need more steps to compile and link them.

The following commands are shown for Arduino Due (note the `cortex-m3` option), other boards will need to have compiler options changes appropriately.

```
$ gcc -c -gnatp -mcpu=cortex-m3 -mthumb pins.adb
$ gcc -c -mcpu=cortex-m3 -mthumb utils.adb
$ gcc -c -mcpu=cortex-m3 -mthumb program.adb
```

You should notice the `-gnatp` option when compiling the `pins.adb` file. This option instructs the compiler not to generate calls to range check functions that verify whether the values that are assigned to variables or converted between types fall within expected ranges and if not, an appropriate standard exception is raised. We have decided not to rely on the Ada run-time library and without this option the final program would not link properly (try it, you can also check the assembly output for the `pins.adb` file and see that indeed there are some calls to other, language-supporting subprograms). Instead, we can verify the code and conclude that all values are safe and therefore resign from run-time range checks altogether, which in the embedded context is a more robust approach anyway. Later on we will see how to automate such analysis, which with bigger programs could be much less straightforward when done manually. For the time being, we will rely on manual verification and the `-gnatp` option in such cases.

Since our program is composed of many files, we need to name them all when linking complete executable:

```
$ ld -T flash.ld -o program.elf pins.o utils.o program.o
```

It is instructive to see what names are defined in the linked program file (shown for Nucleo-32):

```
$ nm program.elf
48000428 A GPIOB_BRR
48000418 A GPIOB_BSRR
48000400 A GPIOB_MODER
080001fc D pins_E
08000100 T pins__enable_output
080001f8 R pins__pin_idN
080001ec R pins__pin_idS
08000160 T pins__write
080001fe D program_E
40021014 A RCC_AHBENR
080001c4 T run
080001fd D utils_E
080001b8 T utils__spin_indefinitely
```

As you see, the symbols describing registers are there as well, in addition to symbols for all of our procedures. For other boards, the symbol names will be of course different and the addresses assigned for each procedure will be different, too.

You can prepare the binary image and upload it to the board as before. You might want to verify that the `run` entry point was automatically reflected in the second word of the binary image, where the reset handler vector is expected. It might have a different value than before, but that does not matter as long as the procedure address is consistent with the vector value - note that the linker takes care of this consistency automatically.

If you run this program you will be able to verify that pin 11 on the board was set to high state (about 3.3V) and pin 12 was set to low state (close to 0V) and you can use these two pins to perform some useful work (taking into account documented load limitations!); other pins have some intermediate voltage with high impedance that makes them effectively disconnected.

Previous: Linking and Booting, next: Very Simple Delays.
See also Table of Contents.