Cours de MICROINFORMATIQUE
Section de Microtechnique
Printemps 2020

Practical Exercises

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

	Practical Exercise 1: Getting Started on e-puck2
Title:	Programming simple functions on the e-puck2 miniature mobile robot.
Goal:	Learn how to program and debug the e-puck2 robot and familiarize with the STM32F4 microcontroller family.
Duration:	4 hours
Support:	Files available to download listed and explained in the appendix.
Equipment:	e-puck2 robot, programming environment tools

1 Introduction

1.1 Some advices for the DLL computers

During all the practical sessions, you will download folders containing the files needed to work and you will have to keep them for the next practical sessions.
Here are some advices to not loose time (and files) :

- Keep everything into the same folder you can call Microinformatique-TPs for exemple.
- Some TPs will use libraries we will provide to you, thus you will have to use your Microinformatique-TPs folder for each TP.
- Do not put your folder on a network drive like the My Document folder. It will be excessively slow to compile since network access has never been fast for accessing a lot of small files. Instead, put your folder on the desktop but don't forget to copy it somewhere at the end of the session, otherwise you will loose it when disconnecting from the computer.
- To copy the folder somewhere, we advice you to use 7zip to compress it. Then copy the zipped version somewhere. On the next session, copy again the zipped version to the computer and unzip it with 7zip. To zip a folder containing a lot of small files and then to copy it is much faster than directly copying the folder, especially with windows.
- You can store the folder on Google Drive, a USB key, etc. Just be sure to have successfully copied it before disconnecting from the computer since as said before, you will loose everything.
- For each PDF you will have to read/use, download it and open it with a PDF reader instead of visualizing inside Firefox for example. This will let you use the table of content of the PDF, thing that Firefox doesn't do well. It is very usefull when you need to find something in the reference manual for example.

1.2 Main Goal

This practical work shows all the necessary steps to program the e-puck2 miniature mobile robot in C, using the standard library provided by ST. The main goal is to gain knowledge of the STM32F4 microcontroller and refresh some concepts about peripherals such as GPIOs and TIMERs.

For this practical work and all the following ones, we supply you with some files that you can download from the course website.

1.3 Methodology

To achieve the main goal, we will go through the following steps:

- Getting familiar with the development environment.
- Getting used to program an e-puck2: use the on board debugger interface for programming and debugging.
- Writing a first LED blinking program, first with NOP loops, then using timers. Understanding how timers can bring precision in timing and how they work.
- Choose a sequence of LED's blinking from the selector

2 Tutorial for programming the e-puck2 robot

2.1 Introduction

The development of embedded systems requires specific programming and debugging tools, allowing to make cross-development, programming on the device and remote debugging. In this tutorial we go through the process of building, programming and debugging code for the e-puck2 miniature mobile robot.

If you are working in the DLL rooms in the MED building, the Eclipse environment is already installed. However if you wish to work using your own laptop or at home, you need to install the Eclipse distribution using the document that you can find in `http://www.gctronic.com/doc/index.php?title=e-puck2_robot_side_development`.

2.2 Download the source for the TP

This practical work will use the standard ST library, which includes the necessary files to program the microcontroller stm32f407 and program it using C language.

Download the zip file containing the sources on the moodle, and unzip it on the desktop. You will obtain the folder **TP1_blinky** which contains two folders: **ST** which contains some standard libraries for ST microcontrollers and **blinky** which contains the source files (**main.c**, **main.h**, **Makefile**, etc.)

Then, open the Eclipse_e-puck2 software and create a new project. For this, click on **File->New->Makefile Project with Existing Code** . Then click on browse and select the **blinky** folder. Choose a name for the project, for instance **TP1_blinky**. Select **None** for the toolchain, click on finish and the project with the source files should appear on the left window.

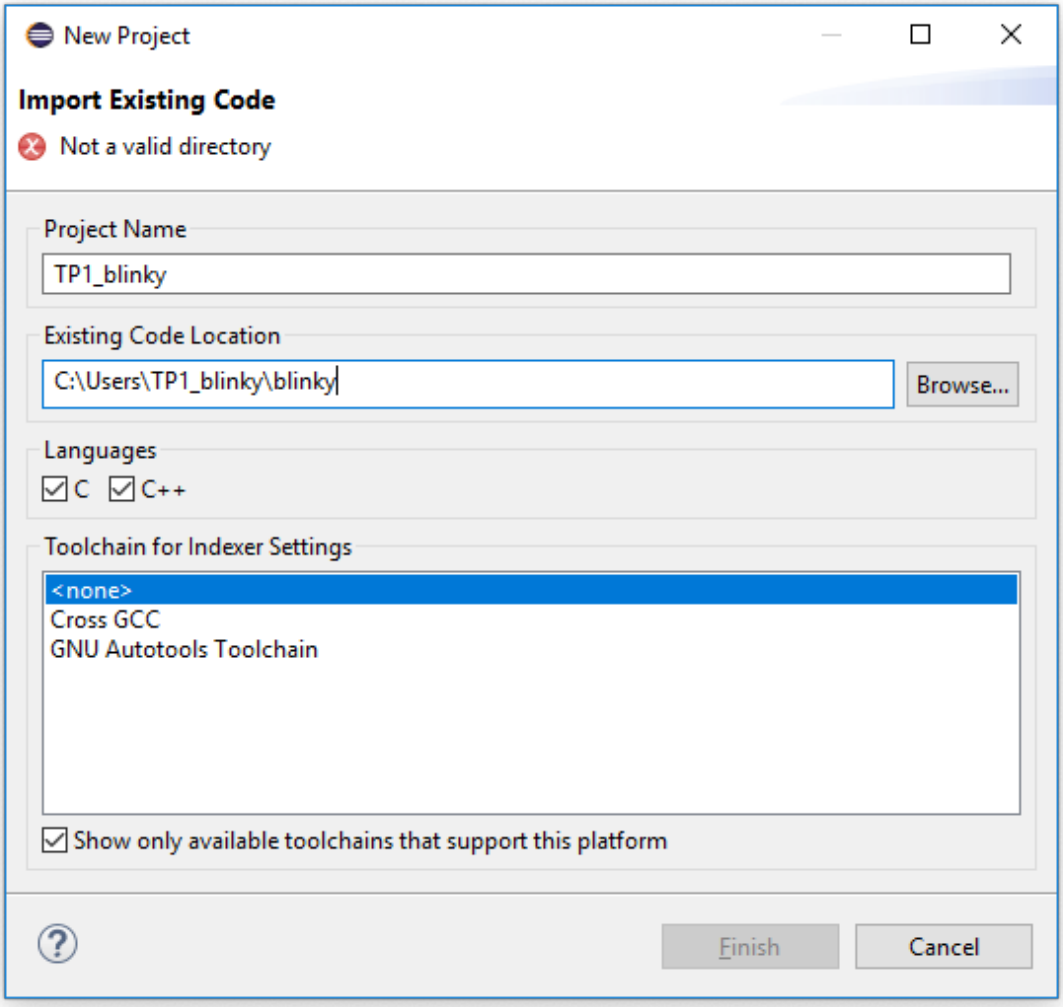

Figure 1: New project Eclipse window

Now you need to add a linked folder pointing to the ST folder. This will let Eclipse know we use files from this folder. This means it will be indexed and we will be able to open the declaration and implementation of the functions contained into the folder. To create the linked folder, go to **File->New->Folder** and click on the **Advanced»** button. Then choose **Link to alternate location (Linked Folder)** and browse to select the **ST** folder and click **Finish**.

Finally, go to **Project->Properties->C/C++ General->Preprocessor Include Paths, Macros etc.->Providers** , check **CDT Cross GCC Built-in Compiler Settings** and finally fill the textbox below with the following command :

Warning : Copy-paste with PDF is not reliable. Double check is the pasted text is correct

```
arm-none-eabi-gcc ${FLAGS} -E -P -v -dD "${INPUTS}"
```

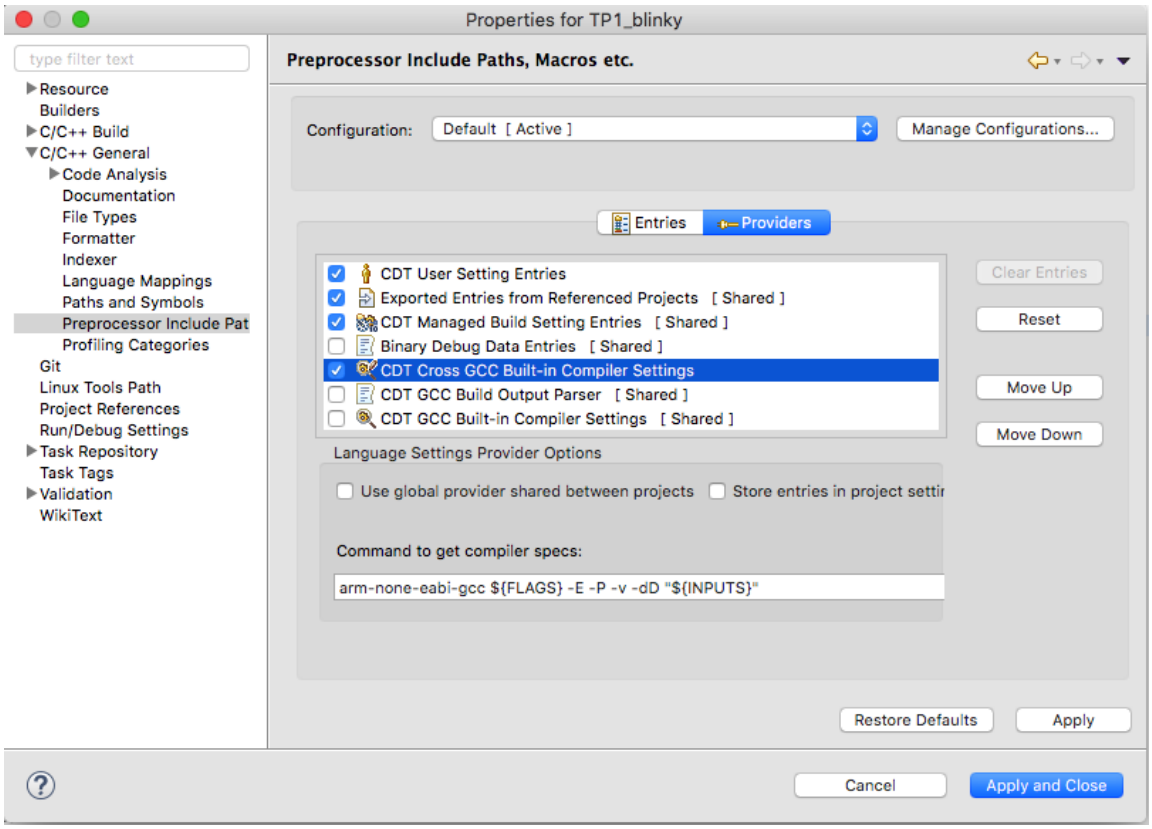

Figure 2: Project's properties window

This last manipulation will let Eclipse know where are the standard libraries of the compiler used (arm-none-eabi-gcc in our case).

2.3 Toolchain and build system

The toolchain to compile source code for the e-puck2 processor is the `GNU ARM Embedded Toolchain`[1]. It includes ports of GNU GCC, binutils and newlib C library for the ARM processor. The build system is a simple Makefile. Open the Makefile of the project to have an overview of the configurations, but do not modify it yet.

The `arm-none-eabi-gcc` executable is used to compile, assemble and link the project.

2.4 Building the Project

Building the code consists of compiling the `*.c` and `*.s` files to create object files, `*.o`, and then linking the object files to create the `blinky.elf` file. We will see in more details in TP2 the different steps of the build process. The generated `.elf` file contains the data necessary to program the device and additional information that lets you debug at the source code level. The additional files `.list`, `.size` and `.mem` are generated containing the disassembly output and the memory layout table with symbol address and size.

Before compiling a specific project, if you have more than one project opened you must be sure to work on the right one, either by having the focus in the editor on a dedicated file of the project, or by selecting any item of the specific project in the Project Explorer tabular.

1. Select **Project>Build Project** in the menu or press on  to start the build.

You can also do a **Right-click** on the specific project in the Project Explorer tabular then **Build Project**.

2. Observe the progress of the compilation in the **Build Console** tabular (as in Listing 1).
3. When **Done** is displayed, your code is built and you are ready to program the device. If there are errors in your code, they will be displayed in this console.

You can also invoke make from the command line to build the project as in Listing 1

Listing 1: Build of the project

```
$ make
make all
> Compiling main.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -O0 -ggdb -fomit-frame-pointer -falign-functions=16 -ffunction-
    sections -fdata-sections -fno-common -Wall -Wextra -Wundef -Wstrict-prototypes -DSTM32F4 -
    DSTM32F407xx -mthumb -mno-thumb-interwork -MD -MP -I. -I./ST main.c -o main.o
> Compiling gpio.c
(...)
> Compiling startup_stm32f407xx.s
arm-none-eabi-gcc -x assembler-with-cpp -c -mcpu=cortex-m4 -mthumb -mno-thumb-interwork -I. -I./ST ST
    /startup_stm32f407xx.s -o ST/startup_stm32f407xx.o
> Linking blinky.elf
arm-none-eabi-gcc main.o gpio.o timer.o ST/system_clock_config.o ST/stm32f4xx_ll_rcc.o ST/
    system_stm32f4xx.o ST/startup_stm32f407xx.o -mcpu=cortex-m4 -O0 -ggdb -fomit-frame-pointer -
    falign-functions=16 -ffunction-sections -fdata-sections -fno-common -nostartfiles  -mthumb -mno-
    thumb-interwork -Wl,--no-warn-mismatch,--gc-sections,--script=ST/STM32F407VGTx_FLASH.ld  -o
    blinky.elf
> Creating blinky.list
arm-none-eabi-objdump -d blinky.elf > blinky.list
> Creating blinky.size
arm-none-eabi-nm --size-sort --print-size blinky.elf > blinky.size
> Creating blinky.mem
arm-none-eabi-nm --numeric-sort --print-size blinky.elf > blinky.mem
arm-none-eabi-size blinky.elf
   text    data     bss     dec     hex filename
   1432       4    1540    2976     ba0 blinky.elf
> Done
```

Home Task 1:

Try to find out the meaning of the words **text**, **data**, **bss**, **dec**. Estimate the size of your code in percentage of total Flash and RAM available with this microcontroller.

2.5 Programming the e-puck2 robot

Now that you have built your code and created an `.elf` file that can be loaded on the microcontroller, you can program the device. To program the robot e-puck2, plug the USB cable and turn on the e-puck2 using the dedicated button.

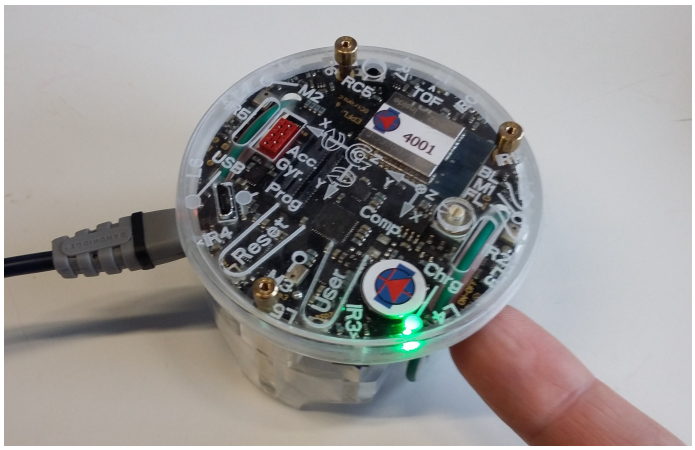

Figure 3: Plug the USB cable using one of the USB connector. Then, turn the device ON by clicking the button located on the bottom surface of the PCB, near the right wheel.

Then, you will need to configure the debugging tool to specify the file you will load on the e-puck2. The e-puck2 robot includes the on-board `Black Magic Probe`[2] debugger. It allows to load, run, halt and step through the code.

Click on the arrow next to the debug button ( ) and select `Debug Configurations`. Then select on the left panel the **Generic Blackmagic Probe** configuration.

Select the **Main** tabular and here, select the project **blinky** and the file `blinky.elf` (Fig. 4).

For this, you have to specify the name of the previously built elf file `blinky.elf`.

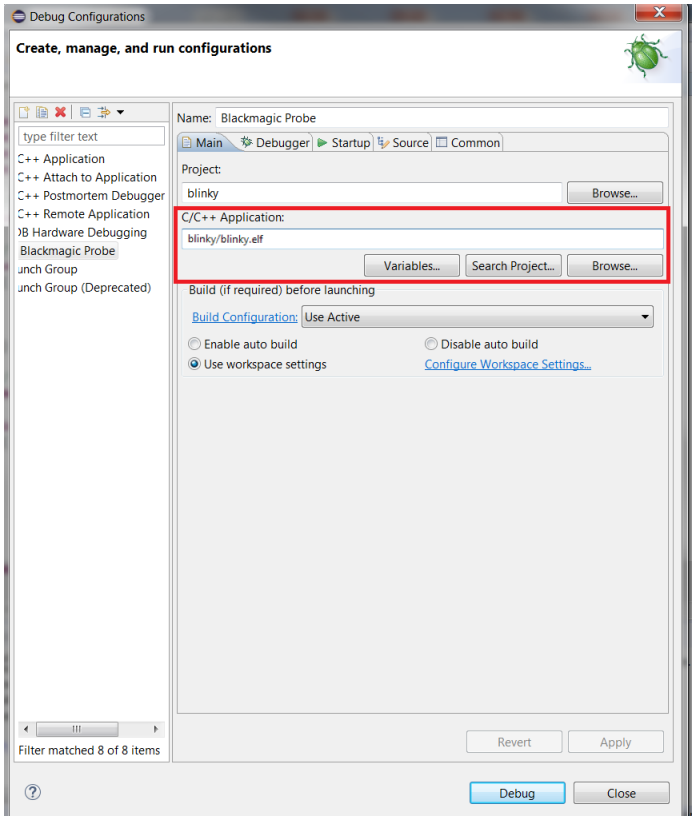

Figure 4: Debug configuration Main

Then, you also have to specify the port on which the Black Magic was installed. For this, first

open the `Device Manager` (**Windows**+**X** then **M** on Windows 10) and identify the ID of the port attributed to the device `e-puck2 GDB Server` (In the case of Fig. 5 it is COM6).

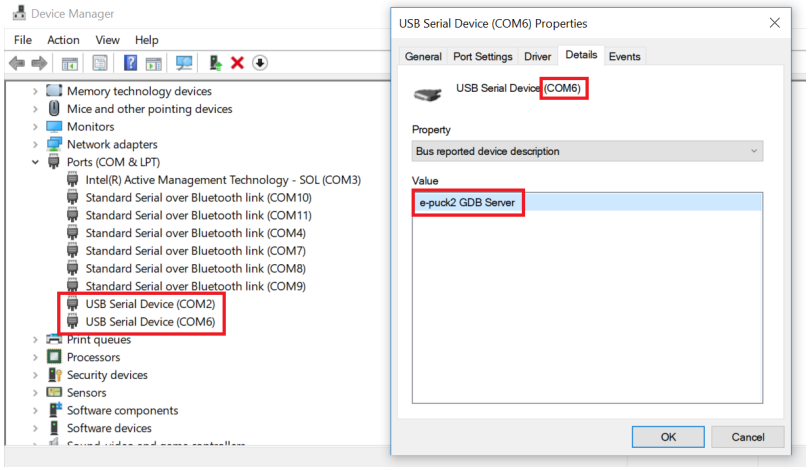

Figure 5: Device Manager

To help you, you can also unplug then plug again the USB cable in order to discriminate the 2 or 3 dedicated serial ports of e-puck2.

Then in the **Startup** tabular of the **Debug Configuration**, write the corresponding COM Port after `target extended-remote` as in Fig. 6.

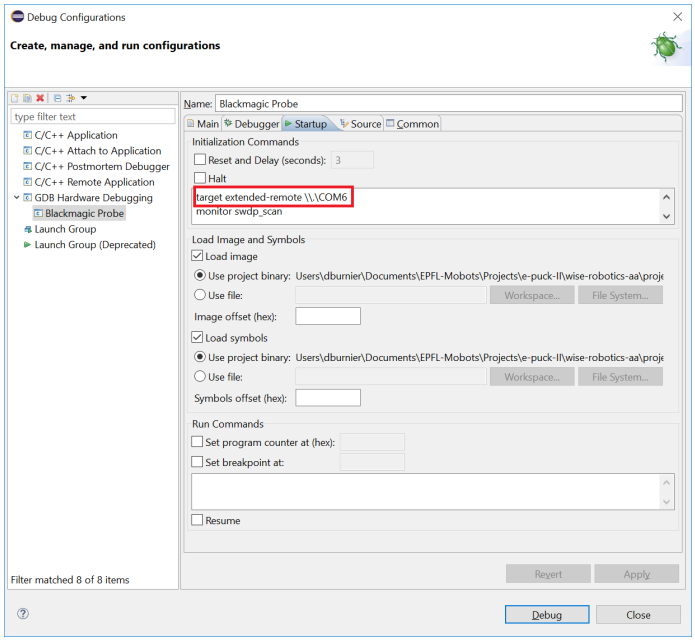

Figure 6: Debug configuration Startup

If you are on another OS (Mac or Linux), you can use the the following chapter of the e-puck2 wiki to help you find the good com port.
`http://www.gctronic.com/doc/index.php?title=e-puck2#Finding_the_USB_serial_ports_used`

2.6 Launch GDB debugger

Before Launching the code, place a breakpoint on the line where you want to stop the program by double clicking on the left of the line number or by selecting the line then *right-click>Toggle breakpoint*.

Click on the arrow next to the debug button () and select your configuration (here **Generic Blackmagic Probe**) to launch the debugger and load the program on the robot.

To step through the code, select the line you want the program to halt then *right-click* on it and select *Run to Line* or *CTRL+R*.

The debugger should stop at the line where you placed the breakpoint. Press  to step line by line.

You can run, pause or stop the code using the buttons  .

WARNING: If you tell the debugger to go to the next step, it will wait until the code goes to the next step. So if you do that on the while loop, it will wait forever and will be blocked. You will have to press the reset button of the e-puck2 to restart the code. Then you will be able to use again the debugger.

To display additional debug information such as disassembly, memory dump and data watchpoints, select for example **Window->Show View->Disassembly**.

When the code is running, the LED 7 of the e-puck2 connected to the pin 11 of the GPIOD should be turned on.

3 STM32F4 Microcontroller and GPIO configuration

To understand the steps necessary to configure a peripheral such as GPIO, we need to have a basic understanding of the system architecture.

3.1 Introduction

As you have seen during the lecture, the STM32F4 is a 32-bit microcontroller series from STMicroelectronics including a standard ARM Cortex-M4 processor core, SRAM and flash memory and various peripherals [3].

3.2 Bus architecture

The STM32F4 main system consists of a 32bit AHB bus matrix which connects the ARM processor to Flash memory, SRAM memory and peripheral buses APB1 and APB2. Figure 7 shows the bus matrix connections in more detail.

The different buses are transparently mapped into the memory address space. For example FLASH starts at 0x08000000, SRAM memory starts at 0x20000000 and peripheral buses AHBx and APBx start at 0x4000000. At the address 0x00000000 is the interrupt vector table which is mapped either to SRAM or FLASH depending on the boot configuration. We will treat interrupts later in this practical exercise in section 4.

After reset, the peripheral clocks are disabled. First, we have to configure the clock and then enable the clock corresponding to the peripheral we would like to configure.

3.2.1 Reminder: Bit manipulation in C.

Listing 2 shows different approaches to manipulate bits in a variable or register using bit masks. [1]

[1] Don't forget & is evaluated before |. Use parentheses to avoid ambiguity.

Figure 1. System architecture for STM32F405xx/07xx and STM32F415xx/17xx devices

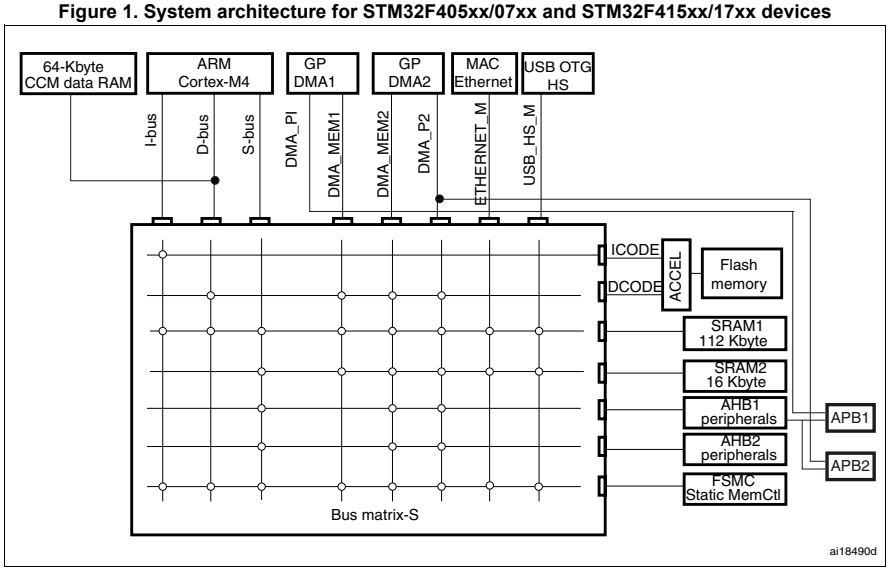

Figure 7: STM32F4 bus matrix, Reference Manual chapter 2 *Memory and bus architecture*[4].

Listing 2: Set and clear bits using bit masks.

```
// set bit 6 in REGISTER
REGISTER = REGISTER | (1 << 6);
// equivalent to
REGISTER = REGISTER | (2 << 5);
// equivalent to
REGISTER |= (1 << 6);
// equivalent to
REGISTER |= (2 << 5);

// clear bits 5 and 6 in REGISTER
REGISTER = REGISTER & ~(3 << 5);
// equivalent
REGISTER &= ~(3 << 5);

// combined
REGISTER = REGISTER & ~(3 << 5) | (1 << 5);
```

3.3 Clock configuration

Open the `main.c` file. In the `main` function, the first instructions consist of calling the function `SystemClock_Config`. Right click on this function and click on `Open Declaration` to open the file `system_clock_config.c` to observe the implementation of this function.

We will not describe in too much details all the calls that are done in this function, however, you can read the comments to have an overview of how the clock is configured (Listing. 3).

Listing 3: Configuration of the system clock.

```
/**
  * @brief  System Clock Configuration
  *         The system Clock is configured as follow :
  *            System Clock source            = PLL (HSE)
  *            SYSCLK(Hz)                     = 168000000
  *            HCLK(Hz)                       = 168000000
  *            AHB Prescaler                  = 1
  *            APB1 Prescaler                 = 4
  *            APB2 Prescaler                 = 2
  *            HSE Frequency(Hz)              = 24000000
  *            PLL_M                          = 8
```

```
  *            PLL_N                          = 336
  *            PLL_P                          = 2
  *            VDD(V)                         = 3.0
  *            Main regulator output voltage  = Scale1 mode
  *            Flash Latency(WS)              = 5
  * @param  None
  * @retval None
  */
```

The system clock is set to 168 MHz. There is a prescaler of 4 configured for the APB1 bus peripheral that will be used further to configure the timer.

Go back to the `main` function in the `main.c` file. After the call of `SystemClock_Config.c`, to use a peripheral, you see that you have to enable the corresponding clock first by setting the enable bit in the registers `RCC_AHBxENR` or `RCC_APBxENR` respectively, depending on which bus the peripheral is connected to.

3.4 GPIO configuration

In this example we configure pin 11 of port GPIOD, which is connected to LED7. To be able to use the GPIOD peripheral we need to enable its clock. All GPIOs are on the AHB1 bus. [2] Here, we enable GPIOD clock by setting `GPIODEN` bit in the `AHB1ENR` register, which is shown in figure 8.

7.3.10 RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Reserved		DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw			rw	rw			rw		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 8: RCC AHB1ENR register in the Reference Manual [4] page 242.

The peripheral registers and bit masks are conveniently defined in `stm32f407xx.h` as shown in listing 4

Listing 4: stm32f407xx.h

```
9904 #define RCC_AHB1ENR_GPIODEN_Pos            (3U)
9905 #define RCC_AHB1ENR_GPIODEN_Msk            (0x1U << RCC_AHB1ENR_GPIODEN_Pos)   /*!< 0x00000008 */
9906 #define RCC_AHB1ENR_GPIODEN                RCC_AHB1ENR_GPIODEN_Msk
```

Register access happens through C structures which are grouped by peripheral. The `RCC` struct groups the *Reset and clock control* and contains the `AHB1ENR`. Finally, listing 5 shows how to enable the clock.

Listing 5: GPIOD clock enable

```
// enable GPIOD clock
RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
```

[2] You can find this information in the Reference Manual [4] or in the datasheet[6] in chapter *Memory map*.

3.4.1 GPIO Peripheral

A General-purpose I/O (GPIO) port has up to 16 I/O pins which can be configured individually. Figure 9 shows the basic GPIO structure for one port bit. Study the schema to get an understanding of the peripheral functions.

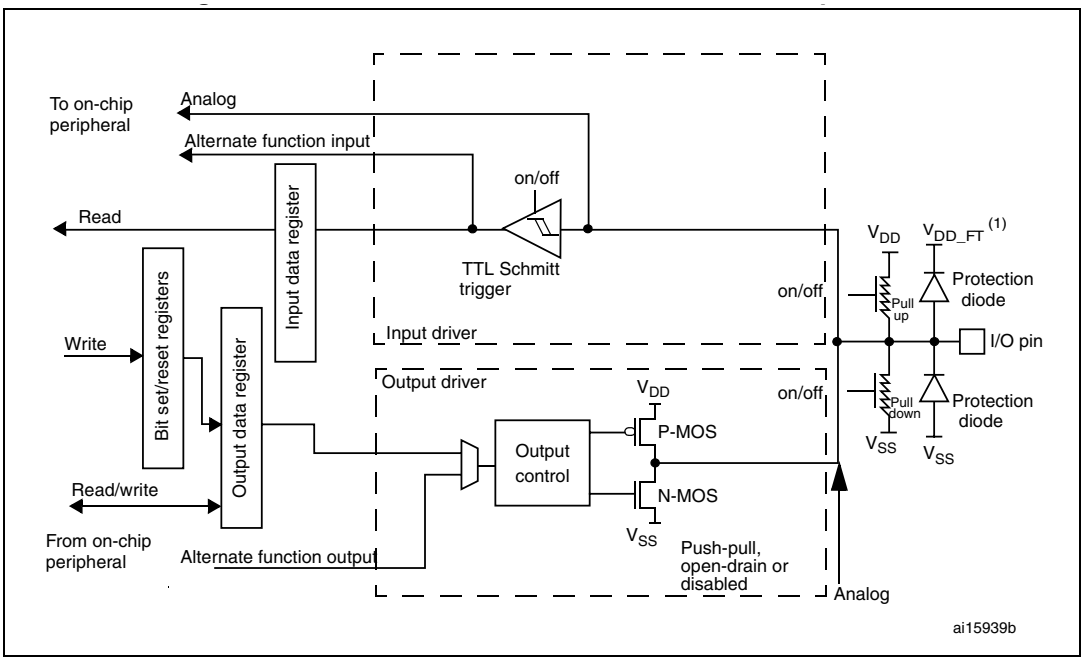

Figure 9: Basic structure of a GPIO port bit.

As you have seen during the lecture, a GPIO pin has 4 configuration registers:

- `MODER`: Mode configuration (input, output, alternate function or analog).
- `OTYPER`: Output type (push-pull or open-drain)
- `PUPDR`: pull type (pull-up, pull-down or floating)
- `OSPEEDR`: output speed (4 levels)

WARNING 1: If you configure the wrong GPIO as output you can damage the robot! So double check before running your code that the configurations are correctly done.

WARNING 2: Remove the battery while manipulating GPIOs to avoid damaging the stepper motors.

Go through chapter 8.4 *GPIO registers* on page 283 of the Reference Manual [4] to find out which bits to set to configure a GPIO pin as open-drain output. Listing 6 shows an example of how to configure LED7 - GPIOD 11 as an open-drain output.

3.5 GPIO data access

There are 3 registers to access the GPIO input and output data:

- `GPIO->IDR` Input data register.
- `GPIO->ODR` Output data register.
- `GPIO->BSRR` Bit set/reset register.

Listing 6: Configure GPIOD pin 11 as output

```
// LED7 - GPIOD 11
#define LED7_PORT   GPIOD
#define LED7_PIN    11

// Enable GPIOD peripheral clock
RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;

// Output mode : MODERy = 01
LED7_PORT->MODER = (LED7_PORT->MODER & ~(3 << (LED7_PIN * 2))) | (1 << (LED7_PIN * 2));
// Output type open-drain : OTy = 1
LED7_PORT->OTYPER |= (1 << LED7_PIN);
// Output data low : ODRy = 0
LED7_PORT->ODR &= ~(1 << LED7_PIN);
// Floating, no pull-up/down : PUPDRy = 00
LED7_PORT->PUPDR &= ~(3 << (LED7_PIN * 2));
// Output highest speed : OSPEEDRy = 11
LED7_PORT->OSPEEDR |= (3 << (LED7_PIN * 2));
```

The 16bit `IDR` register allows reading GPIO input state and the 16bit `ODR` register allows writing GPIO output state.

To change the output state of a pin you have to read the register value, apply a bit mask and write it back to the register. This can be problematic because between read and write an interrupt can occur and the same register might be accessed form the interrupt service routine. In this case the change from the interrupt will be overwritten by the write back of the base program.

To address this problem there is a third 32bit register, the `BSRR`, which is write-only. It allows setting and resetting `ODR` bits by one write access. Writing a one to the upper 16 bits will reset the corresponding bit in `ODR` to 0, while writing a one to the lower 16 bits will set the corresponding bit to 1.

3.6 Observe the registers during debug

It is possible to watch the state of the register when the code is running on the microcontroller. We will do it with the simple example code (Fig. 10).

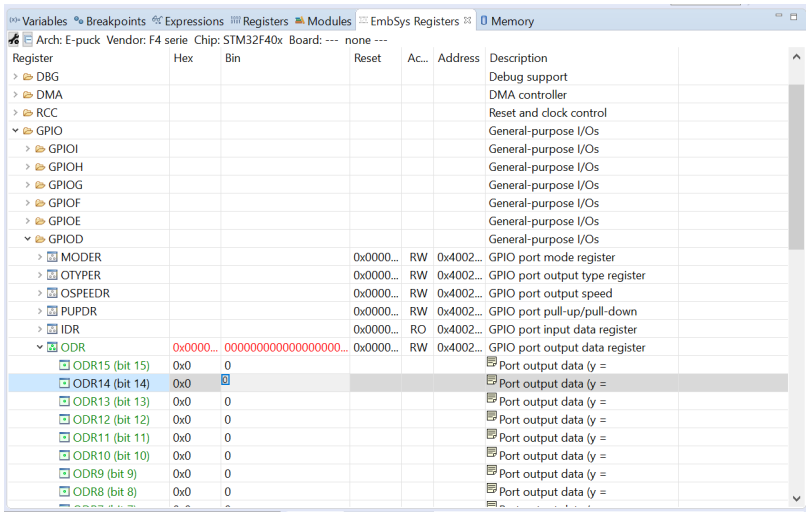

Figure 10: Visualization of the registers during debug.

Build your code and launch the debugger. On the top-right window of the Eclipse workspace, click on the `EmbSys Registers` tabular.

Then, open the folder `GPIO`, `GPIOD`, and `ODR` to see the output register status. Double click on the `ODR` folder to activate the actualization of the registers inside this folder (marked in green). The Debugger should be in pause or halted at a breakpoint to actualize the values. When you click on the `Bin` column for the `ODR11`, try to change the value from 0 to 1, and click `Set`, the LED of the E-puck2 should turn off.

3.7 Blink a LED automatically

Task 1:

Use a simple delay function to make **LED7** blink in a while loop at 1 Hz.

Hints:

- Consult the electrical schema of the e-puck2 to find the LED7 IO pin.
- Use the functions declared in *TP1_ blinky/blinky/gpio.c* to modify the state of the pin on which the LED is connected, and create a function that generate a time delay using assembly *nop* instructions.

3.8 Blink only the Front LED

Task 2:

Modify your code to blink only the 5mm red **FRONT_LED**.

Hints:

- Consult the electrical schema of the e-puck2 to find the FRONT_LED IO pin. If the LED doesn't blink, compare the LED driving topology with the previous one and try to play with the **PUPDR** and **OTYPER** GPIO registers from the *EmbSys Registers* tabular.
- Use the oscilloscope to measure the voltage level on the FRONT_LED test point, under the marked **FL** on the transparent cover (Fig. 11). **You can connect the GND to the screw that holds the cover.**
- Describe the influence of both registers and try to explain the situation. Adapt your code for this *Output topology*.
- Take the opportunity to use the oscilloscope to measure the influence of 2 extreme values of **OSPEEDR** configuration for this GPIO and take a plot of both for rising and falling edges.

[3]

3.9 Blink only the 4 Body LEDs

Task 3:

Modify your code to blink only the 4 green **BODY_LEDs**.

Hints:

- Consult the electrical schema of the e-puck2 to find the BODY_LED IO pin.
- If the LEDs don't blink, check in detail all characteristics of this GPIO (port, pins, topology).
- Can you explain why this output topology is used for BODY_LED and not the same than the LED7 one for example?

[3] Don't forget to pause the debugger to be able to change the register value from the *EmbSys Registers*.

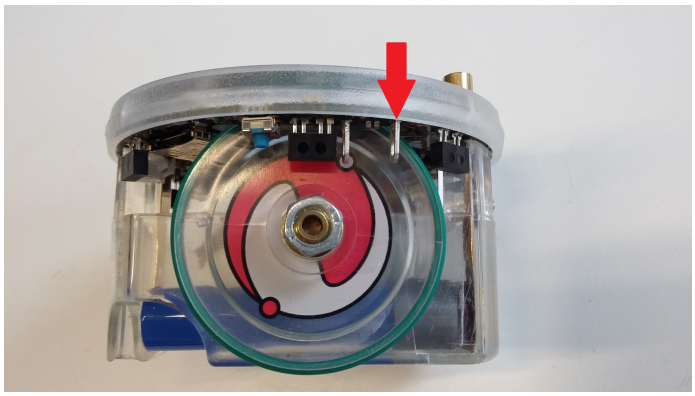

Figure 11: Connect the oscilloscope to the test point that is linked with the FRONT_LED.

3.10 Blink many LEDs automatically with a circular pattern

Task 4:

Blink the LED1 -> LED3 -> LED5 -> LED7 with a circular pattern, with ON1-ON3-ON5-ON7-OFF1-OFF3-OFF5-OFF7 or ON1-OFF1-ON3-OFF3-ON5-OFF5-ON7-OFF7 sequence depending of the Selector state.

Hints:

- Consult the electrical schema of the e-puck2 to find the LED1, LED3, LED5 and the Selector IO pins.
- Create a function that returns the Selector states. You must obviously configure the correct pins in the correct way to be able to read the Selector state. You can then visualize the state of the Selector by looking at the IDR register of the related pins with the *EmbSys Registers* tabular.
- Have a look on the comment close to the selector on the schema. Can you explain it?
- Define different LED sequences depending on the Selector state.

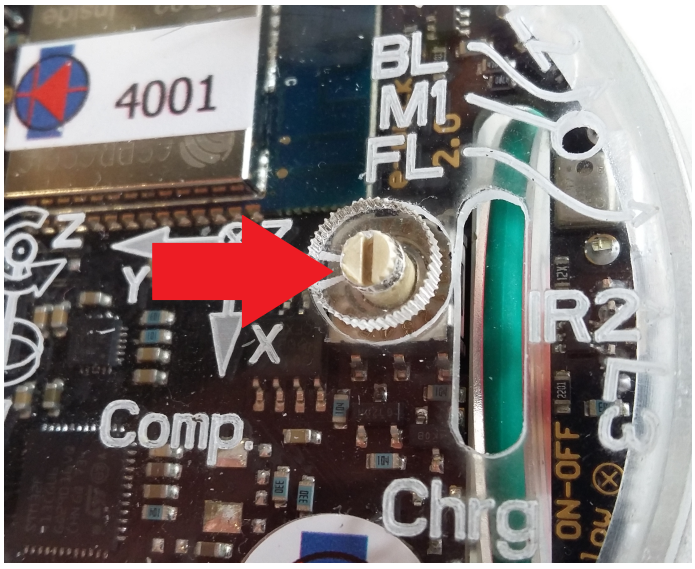

Figure 12: The e-puck2 is equipped with a selector that you can use to configure different modes.

4 Timer Interrupt Tutorial

In this section, you will use a timer to make the LED blink instead of using a while loop as you did in the previous exercise. To do so, we ask you to configure the timer **TIM7** to periodically count up and, when it reaches the value set in the **ARR** register, to update the counter with a reload value and generate an interrupt. It is then in the interrupt routine generated by the timer that you will have to change the state of the LED.

4.1 STM32 Timer configuration

The configuration of the timer will be done in the function `timer7_start` that is implemented in the file `TP1_blinky/blinky/timer.c`. Go to this file and observe how the timer should be configured (Listing 7).

Listing 7: Configure TIM7 for periodic interrupt

```
void timer7_start(void)
{
    // Enable TIM7 clock
    RCC->APB1ENR |= RCC_APB1ENR_TIM7EN;

    // Enable TIM7 interrupt vector
    NVIC_EnableIRQ(TIM7_IRQn);

    // Configure TIM7
    TIM7->PSC = PRESCALER - 1;      // Note: final timer clock  = timer clock / (prescaler + 1)
    TIM7->ARR = COUNTER_MAX - 1;    // Note: timer reload takes 1 cycle, thus -1
    TIM7->DIER |= TIM_DIER_UIE;     // Enable update interrupt
    TIM7->CR1 |= TIM_CR1_CEN;       // Enable timer
}

// Timer 7 Interrupt Service Routine
void TIM7_IRQHandler(void)
{
    /*
    *
    *   BEWARE !!
    *   Based on STM32F40x and STM32F41x Errata sheet - 2.1.13 Delay after an RCC peripheral clock
     enabling
    *
    *   As there can be a delay between the instruction of clearing of the IF (Interrupt Flag) of
     corresponding register (named here CR) and
    *   the effective peripheral IF clearing bit there is a risk to enter again in the interrupt if
     the clearing is done at the end of ISR.
    *
    *   As tested, only the workaround 3 is working well, then read back of CR must be done before
     leaving the ISR
    *
    */

    /* do something ... */

    // Clear interrupt flag
    TIM7->SR &= ~TIM_SR_UIF;
    TIM7->SR; // Read back in order to ensure the effective IF clearing
}
```

Before you can access the timer registers you have to enable the clock. TIM7 is connected to APB1 as it is shown for instance in the block diagram of the microcontroller in the datasheet, so you have to set the `TIM7EN` enable bit in `RCC->APB1ENR`.

Every interrupt must be enabled in the *Nested vectored interrupt controller (NVIC)* in order to generate an interrupt. To do this just call the provided function `NVIC_EnableIRQ(TIM7_IRQn)` with the interrupt number as argument, in this case `TIM7_IRQn` (IRQ stands for Interrupt ReQuest). The interrupt number is defined in the enum `IRQn_Type` in the header file `stm32f407xx.h`.

Then, you need to implement the interrupt service routine (ISR) which is a predefined function, in this case `void TIM7_IRQHandler(void)`. The address of the interrupt routines is placed in the interrupt vector table, which is defined in the file `TP1_blinky/ST/startup_stm32f407xx.s`. By implementing the interrupt handler the default handler is overwritten.

Now you can start configuring the TIM7 timer using following registers:

- `CR1`: Control Register 1, configure and enable the timer.
- `DIER`: DMA/Interrupt Enable Register.
- `PSC`: Prescaler Register, change the counting frequency by dividing the base clock.
- `ARR`: Auto-Reload Register, value at which the timer will reset its counter to 0 and generate an interrupt.

4.2 Configure TIM7 to have interrupts at 1Hz

Task 5:

Look through the chapter 20 *Basic timers (TIM6 and TIM7)* in the Reference Manual[4] to understand how to configure the TIM7 timer. Configure TIM7 for an interrupt frequency of 1Hz. To do this you have to choose the correct timer prescaler **PSC** and reload value **ARR**.

Hint:
Find out the frequency of the bus on which the TIM7 is connected and deduce the frequency of the timer.

4.3 Interrupt routine

Now, when the timer update event triggers the interrupt, the execution of the main program is halted, the processor's registers are saved on the stack and the processor's execution jumps to the corresponding address in the interrupt vector table. When the ISR returns, the processor's registers are restored and the execution of the main program continues.

In the timer interrupt you have to manually clear the update interrupt flag `UIF` in the timer status register `SR` (see *TIM6/TIM7 status register (TIMx_SR)* Reference Manual [4] page 708).

4.4 Toogle LED7 with TIM7 at 1Hz

Task 6:

Write a code that toggles the LED 7 of the e-puck2 robot using timer 7 implemented in the file *timer.c*. Include the *timer.h* header in the *main.c* file. Build and run your code. Verify that the interrupt frequency is correct by blinking the LED7.

References

[1] *GNU ARM Embedded Toolchain* GCC ARM Embedded Maintainers. [Online; accessed 8 June 2017].
`https://launchpad.net/gcc-arm-embedded`

[2] *Black Magic Probe* Black Sphere Technologies. [Online; accessed 8 June 2017].
`https://github.com/blacksphere/blackmagic/wiki`

[3] *STM32 32-bit ARM Cortex MCUs* STMicroelectronics. [Online; accessed 5 June 2017].
`http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html`

[4] *Reference Manual, STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs* RM0090 Rev 14. STMicroelectronics. September 2016, `http://www.st.com/resource/en/reference_manual/dm00031020.pdf`

[5] *User manual, Discovery kit with STM32F407VG MCU* UM1472 Rev 6. STMicroelectronics. May 2017.
`http://www.st.com/resource/en/user_manual/dm00039084.pdf`

[6] *Datasheet STM32F407xx* DocID022152 Rev8. STMicroelectronics. September 2016.
`http://www.st.com/resource/en/datasheet/stm32f405rg.pdf`