




Hands-On: MCU operation and baremetal programming

Introduction

In this Hands-On, you will be introduced to the basics of microcontroller unit (MCU) operation and interrupt-based embedded programming. We will use so-called bare metal programming, which means we will not use an operating system: you will be in control of all the code that runs on the device! As this is most likely your first project dealing with bare metal embedded programming, we will provide you with a functional baseline code to start with. Then, you will be guided step by step to develop the key skills needed for this part of the project.

Explanation of the hands-on boxes

In this note, there are a few boxes presenting additional information:

-  will provide you some more detailed explanations.
-  will explain typical mistakes that might lead to errors or a non functional system.
-  will provide you with additional questions or experiments that will improve your understanding of the system. We advise you to leave them for the end of the hands-on as they are not critical.

Objectives

- Discover the Nucleo MCU board and its programming tools ;
- Learn the basics of bare metal embedded programming, General-Purpose Inputs-Outputs (GPIOs), interrupts and timers ;
- Understand qualitatively and measure quantitatively how the embedded software program impacts the MCU power consumption ;
- Learn how to debug an embedded program by logging/tracing and using a physical debugger.

Material

For this hands-on session, you will need:

- The latest version of STM32 Cube IDE installed on your host system (for further information, see INSTALL.md on the GitHub of the course) ;
- The Nucleo MCU board with its sensing and power-management board ;
- A USB cable to connect your computer to the Nucleo.

1 Practical part

1.1 Starting from the baseline version

As mentioned in the introduction, you will not start from scratch. We provide you with a baseline version of the code you need for this Hands-On. To ensure you have the latest version of the code, perform a `git pull` on the GitHub project repository and then open STM32CubeIDE. At the first launch of the IDE, it will prompt you to choose a workspace, you can either keep the default one or choose an alternative folder. In order to open the project in the IDE, click on **File** → **Open Projects from File System**. Then in the new window, click on **Directory** to choose the `mcu\hands_on\mcu` folder. Make sure that a project appears in the box below, select it if not selected. It should look as in [Figure 1](#). Click on **Finish**. On your left, browse your source files, your main function will be in **Core** → **Src** → `main.c`, if you open it, your IDE should look like [Figure 2](#).

Now, in order to compile the binaries for the MCU, you need to first download the software and firmware packages of the MCU. At one point of the following procedure, CubeIDE might ask you to identify with an ST account, either log in or register if not done yet. On top of the STM32CubeIDE window, click on **Help** → **Manage Embedded Software Packages**. In the "STM32Cube MCU Packages" tab, search for "STM32L4", check the latest package (normally 1.18.0) and perform the installation. Now, open the `hands_on_mcu.ioc` file, which purpose will be detailed later. If you cannot open the `ioc` file due to an older version of STM32Cube, please follow the displayed guidelines to perform the software updates. Finally, on the main window click on **Project** → **Generate code**. Wait for the download to be finished.

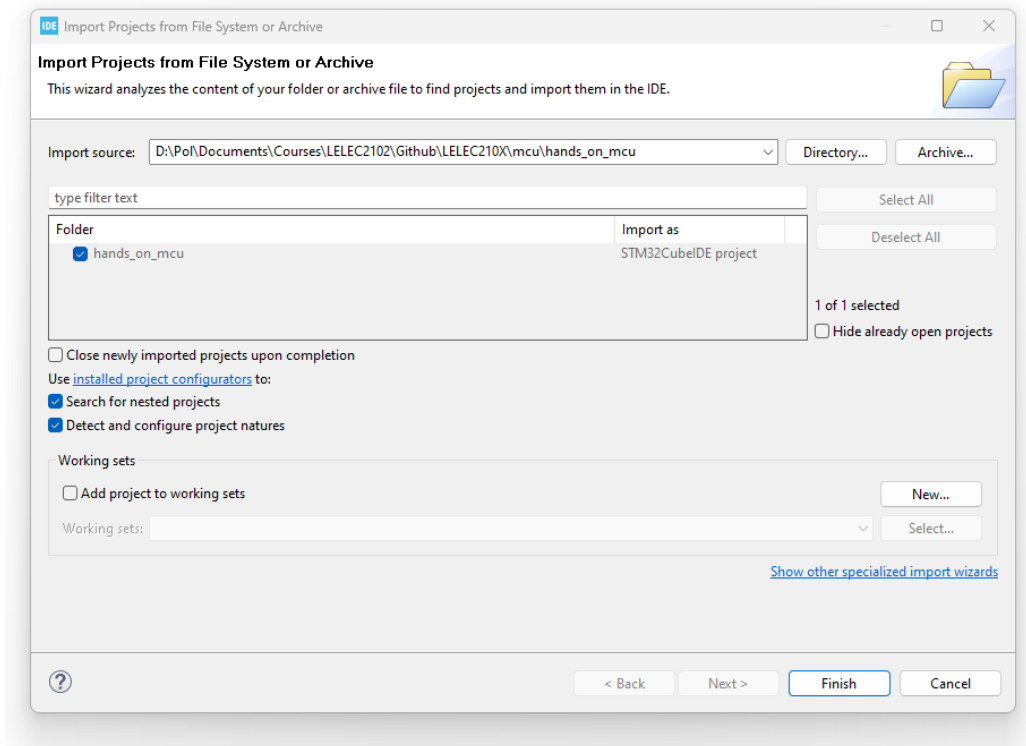


Fig. 1: IDE open project from file system window

You can now flash the MCU (i.e. program its Flash memory) with the baseline version of the code by using the USB cable. This is the first step that makes sure everything is working as expected. To do so, click on the button surrounded by the red circle in [Figure 2](#). The first time you launch a project, the IDE will prompt you the launch configuration. If you followed the tutorial correctly, the default options should fit your needs and you can directly click on **OK**. An error might appear on

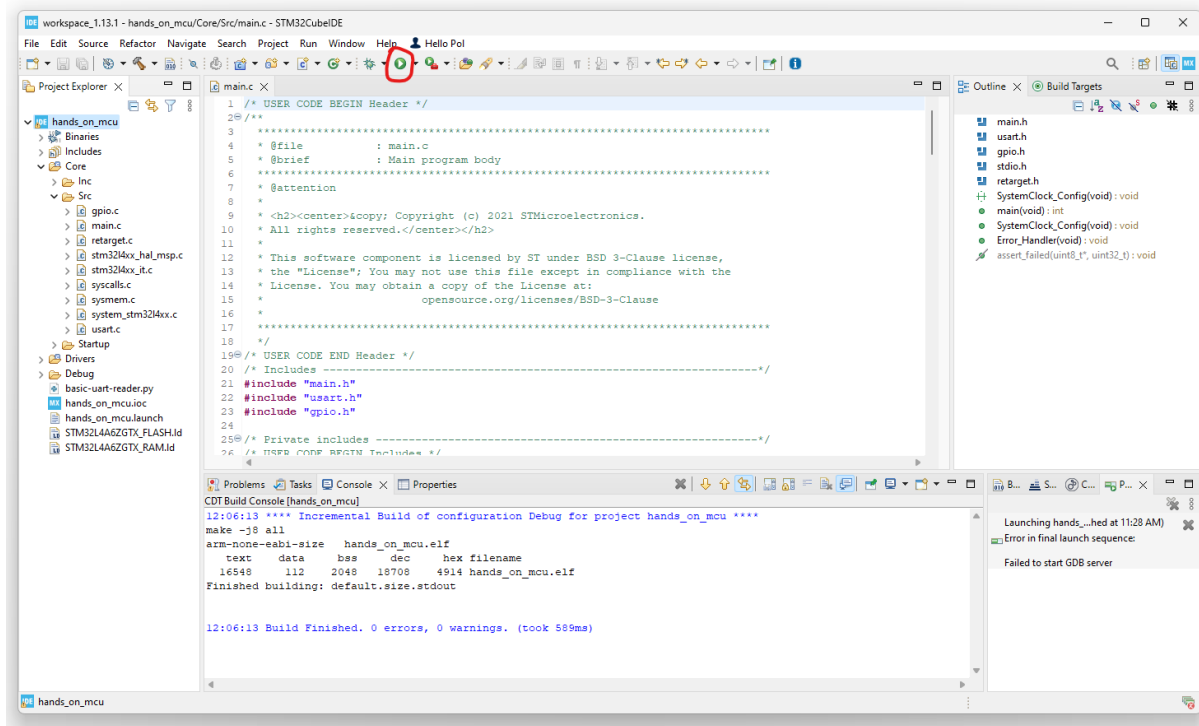


Fig. 2: STM32Cube IDE

your screen, with another message specifying that an ST-Link firmware upgrade is required. If it is the case, click on **Yes**, perform the upgrade and try to reflash the MCU. The baseline version of the code should now be flashed on your MCU and it should be running properly: *check that a blue LED is blinking when you hold the blue button on the Nucleo MCU board.*

ST-Link programmer

The **ST-Link** is a programmer and debugger. It provides the connection between your computer and the MCU on the board. It can be connected to your MCU with a connection called JTAG. In the case of the Nucleo MCU board, a dedicated **ST-Link** is soldered onto the board and is always connected to the MCU. It serves also as a USB to UART adapter.

In order to read the messages that are sent from the MCU through the UART interface, you can open a terminal and launch the python script `basic-uart-reader.py`. It should display the available communication ports. Plug and unplug your Nucleo while launching the script to identify its corresponding port. You can now relaunch the script while specifying the port, e.g., `python3 basic-uart-reader.py -p PORTNAME`. You should be able to see the messages coming from the UART. Press the black reset button to reset the MCU so that it sends "Hello World!" again.

Serial communication baudrate

If you inspect `basic-uart-reader.py`, you will see that a parameter `baudrate` is specified when opening the communication. The **baudrate** of a serial connection represents **number of bits per second that are sent over the connection**. This parameter must be identical between the transmitter (here the MCU) and the receiver (here the ST-link programmer on the Nucleo board that forwards the messages to your computer through USB).

Redirecting stdout to UART

You can use the usual `printf` function family to print information via the UART stream. Note that this is not the case by default if you start a new project from scratch. For your information, the code that redirects stdout to the UART stream lies in `retarget.c`.

1.2 Using a state variable

Have a look at the `main()` function and analyze its behavior. In particular, be sure that you understand the purpose of the HAL functions that are called.

Hardware Abstraction Layer (HAL)

The MCU and its peripherals are controlled by memory-mapped registers. That is, in order to control them, one has to perform a **read/write at a specific memory address** (the address and value written determine the action performed – you will find these in the reference manual of the MCU). ST provides a HAL, which is a set of functions that abstract sequence of such reads and writes. We strongly recommend you to use it (see the HAL description document). Moreover, you need to know that there are two kind of HAL functions provided: the standard ones and the low level (LL) ones. The standard ones are easier to use than the LL ones. For this reason, we recommend you to stick to the standard HAL functions (without `"_LL_"`).

Let us now modify the LED behavior: we want the LED to toggle between blinking and no-blinking modes each time we press the button. Implement this using two boolean variables to store the current state of the system (button pressed or not, LED blinking or not). Then, flash the new version of your code and check that it is working as expected !

STM32CubeIDE code generator

Some of the code in `main.c` has been auto-generated by the IDE, and will be overwritten (soon) when we will change some configuration parameters. When modifying your code, make sure to modify only the code that is between a `USER CODE BEGIN` and `END` (do not modify these comments). These parts of the code will not be overwritten, therefore preserving the changes you have implemented.

```

    /* USER CODE BEGIN xxx */
    // insert your code here
    /* USER CODE END xxx */

```

If your project grows larger, you can also add your own `.c/.h` files (there is a menu for that). They are not overwritten by the code generator, except if there are collisions in the file names you and the code generator want to use. We therefore advise you to commit/backup your changes before running the code generator.

1.3 Interrupt-based programming

As your code is implemented using a polling strategy (i.e., the state of the system is regularly checked, and appropriate action is taken). This approach has several drawbacks, one of them being the complexity to handle multiple tasks at the same time. For instance, if you quickly press the button when your code is executing a delay with the `HAL_Delay()` function, the button press can not be detected. While this limitation can be fixed, polling multiple peripherals and running multiple time-delayed actions in parallel will quickly make your code very complex and most likely less robust to unexpected behaviors.

You will now switch your software to an **interrupt-based implementation**. As seen in the L2 lecture, **the interrupt stops the main activity of the processor and switches to the action to be done when a specific event is detected**. In our case, we will set up the MCU such that **on a rising edge on the button GPIO**, the MCU enters in a function where we can set the blinking state of our system.

Interrupts terminology

An interrupt originates from an **interrupt request (IRQ)** which can originate from outside the MCU (typically an event on a GPIO pin), from an MCU peripheral (e.g. a timer) or from the software (by writing to a specific register). The IRQ is handled by the interrupt controller (a hardware component of the MCU), which then runs the interrupt handler, also named interrupt service routine (ISR), or callback (this is the name used by the ST HAL).

To do so, we first need to set the GPIO pin linked to the button in interrupt mode. The IDE provides an easy way to this, in your source files, open the file `hands_on_mcu.ioc`. It is a configuration file for all the parameters of the MCU, peripherals, clock tree and project. If it is not your case by default, open the **Pinout & Configuration** tab. On the left of the screen, you see all the available peripherals on the MCU. You can click on a peripheral to open a panel where you can set its parameters. In the main panel, you see the different pins of the MCU and their modes. Here, you can click on each pin and set its functionality.

Find the pin PC13 connected to the blue button. You can observe that it is set as GPIO input. To be able to trigger interrupts, we need to use the **GPIO EXTERNAL Interrupt mode: GPIO_EXTI**. Select it when clicking on the pin as illustrated in **Figure 3**. Note that the name of the pin changed. Change it back to B1 by right-clicking on the pin. Now, we need to set up the parameters of Nested Vector Interrupt Controller (NVIC) such that it processes the signal sent by the GPIO pin. To do so, click on NVIC under System Core in the left panel. You will see a panel appear as in **Figure 4**. Here you can select which interrupts you want to enable or not. Some of the interrupts cannot be disabled. You need to activate here the EXTI line. That's all we need to do in the parameters. You can save the `.ioc` file, the IDE will prompt you to regenerate the code based on your modifications, answer yes (remember: that is the reason why you need to write your custom code in the USER CODE areas!).

GPIO Interrupt trigger parameters

By default, the MCU is configured such that a rising edge of the GPIO pin triggers an interrupt request (IRQ) to the CPU. This corresponds to pressing the button (checkout the schematic of the MCU board to understand why!). It is possible to configure it for alternate events, such as a falling edge, corresponding to the release of the button. To do so, open the parameters of the GPIO peripheral under System core in the `.ioc` file, select the GPIO tab in the parameter panel and select the pin that you want to configure. You can change the edge detection in the GPIO mode setting in the lower part of the screen.

When an IRQ is triggered, the NVIC makes the CPU call and interrupt handler function. With the ST HAL, you can write code to be executed in the handler in a "callback" function. In your main file, insert the EXTI interrupt callback function as below.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == B1_Pin) {
        // Insert code to be executed in the interrupt
    }
}
```

check quel
pin a été
trigger

This function will be called each time a GPIO EXTI interrupt is raised, whatever the concerned pin. This is why we need the if condition to check which pin was triggered. Now, remove the GPIO read function from your main loop and update your state variable in the callback function of the interrupt handler.

⚠ Data races and volatile variables in C

Your state variable will be a global variable, such that it can be modified in the main function and in the interrupt handler. The main function and the interrupt handler are concurrent: they are not part of the same “execution flow”, similarly to threads that run under an OS. With all concurrent programming, you have to be careful the access to variables that are shared (i.e., read or written from more than one execution flow). If at least one of the execution flow writes to a variable (**concurrent reads are always ok!**), the result of the execution might be non-deterministic (e.g., depend on the exact way the execution flows are interleaved), which is called a data race. For instance, if you increment a variable (which makes 3 instructions: load, add and store) both from `main()` and from an interrupt handler, you have a data race if you take no extra precaution: if the `main()` is interrupted just after the load, the variable gets incremented by only 1 instead of 2. In order to avoid data races, all concurrent read and writes should be either atomic (roughly, a single load/store) or synchronized (through locks, etc.).

This is further complicated by the semantics of the C programming language: it is such that the compiler can (almost) assume that there is no concurrent execution. Therefore, if a program does not read a variable from the `main()` execution flow, it may “optimize” your `main()` by discarding writes to this variable, as they are “useless”.

As a result, whenever a variable is **shared** (i.e., accessed by multiple execution flows), or represents a **memory-mapped control register**, you have to make it **volatile**. This is done by putting the qualifier `volatile` before the type in the declaration of the variable.

This was only a very limited introduction! To go further: <https://blog.regehr.org/archives/28>

You may also search the keywords: C memory model; atomic memory access ordering; memory barrier.

ℹ Interrupt callback good practice

Be careful not to lock-up your processor: if you have a long-running interrupt handler, the MCU might not be able to respond to other events that happen. The two common solutions are to delegate the long processing to the `main()` (and communicate that the processing should be performed by setting a state variable in the interrupt handler), or to configure the priority of the long-running interrupt such that it can be interrupted by other, higher priority, interrupts. Be careful as usually, the highest priority has the lowest value i.e. 0 being the highest priority as shown in **Figure 4**.

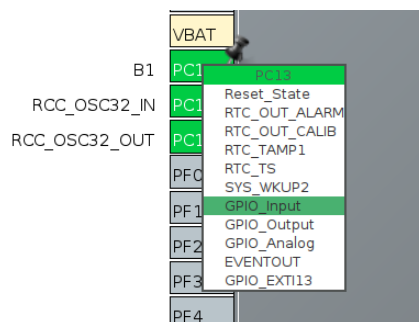


Fig. 3: Cube MX GPIO pin mode selection

1.4 Using a TIMER to blink a LED

Now that you have a code that uses interrupts to detect a press on the button, you will use a more advanced way to blink the LED. **Imagine your MCU has to blink its LED at a very specific frequency**

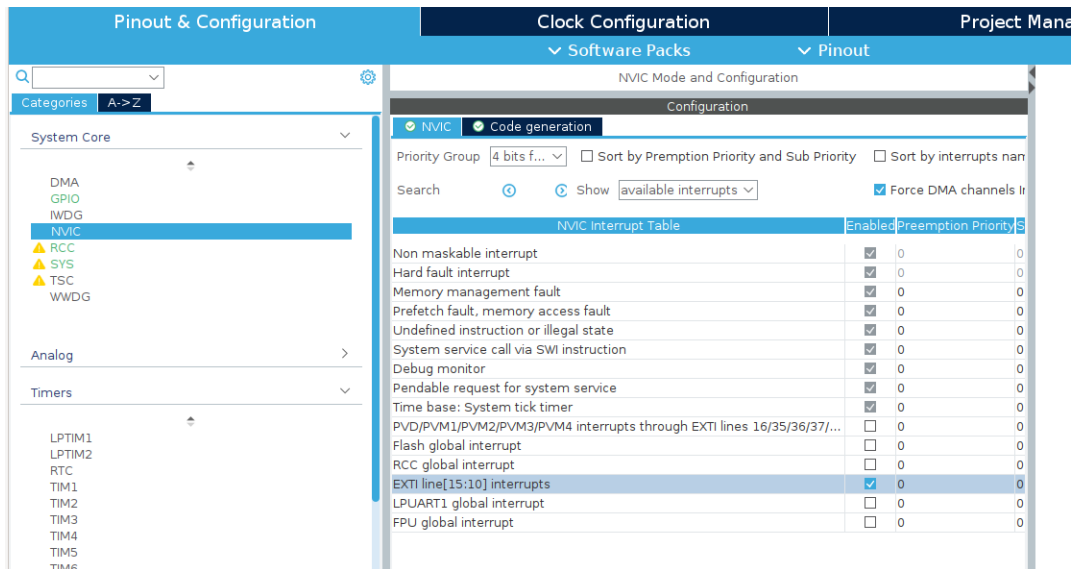


Fig. 4: NVIC parameter selection

and must at the same time, perform heavy calculations. In such cases, the blinking might be altered due to the computation time of the other tasks of the MCU. This can be avoided by using a peripheral of the MCU to move the task of blinking from the MCU to this peripheral called **timer**.

Timers are simple circuits that increment a register by 1 at each clock cycle and perform additional actions when certain conditions are met. Such actions can be reading or writing to a GPIO pin, triggering interrupts and more. A typical condition for a timer is **comparing** the value of the counter to another (fixed) value, and triggering an action whenever the two values are equal.

We will now show you how to setup the timer. A timer has multiple channels that can each perform a condition-action pair. Each channel can be linked to at most one of the few GPIO pins that it is connected to. On the device you have for the project, there are several timers available. For determining which timer channels are linked to which GPIO pins, you can take a look at the datasheet of the MCU. In our case, the blue LED is connected to PB7, which can be connected the channel 2 of timer 4. To do so, open the .ioc file and the **Pinout & configuration** tab and look for pin PB7. When clicking on it, choose **TIM4_CH2** as in Figure 5. Still in the .ioc file, we will now enable

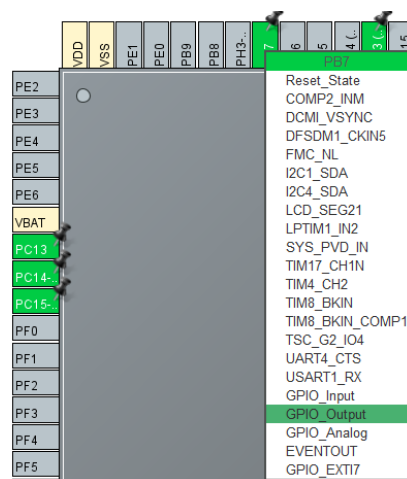


Fig. 5: Possible uses for pin PB7

the timer **TIM4**. Click on the name of this peripheral in the leftmost column of the .ioc window, in the submenu **Timers**. You should see all the parameters of TIM4 in a new panel. In the **mode** panel, you should set the clock source to **Internal clock** and channel 2 to **Output compare CH2**, the rest

should remain disabled. In the configuration panel, you should set the **Prescaler, Counter Mode, Counter Period and Mode** as in Figure 6

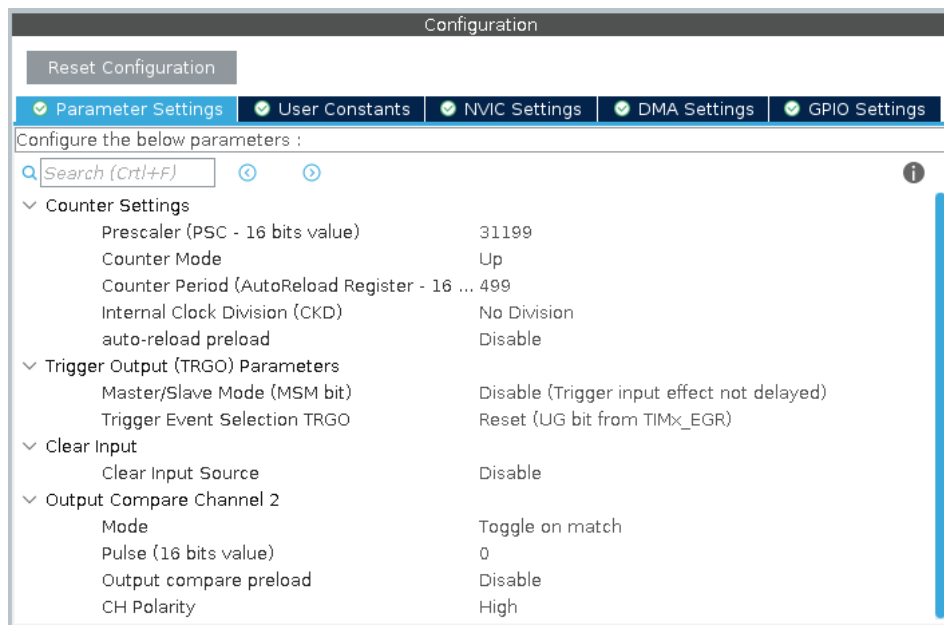


Fig. 6: Timer settings

? Timer parameters analysis

- Take a look at the mode and the proposed values. Try to calculate the frequency of the blinking without launching the code. (Hint: start by checking what is the clock frequency feeding your timer peripheral in the **Clock Configuration** panel of the .ioc).
- Now, change the Pulse value to anything between 0 and the counter period and describe what happens.
- What happens when you set a pulse value above the counter period?

↳ ne s'allume pas

Timer parameters detailed

- Take as example an MCU clock that runs at 8MHz the timer triggers an event every second, this means that a 16-bit timer is not suitable as the compare register should be set at 8000000, but it's maximal value is 65535. This is why we have **prescalers**, which make the timer counter register increment every prescaler value cycles. Therefore, the update frequency f_{update} is given by $f_{update}[Hz] = \frac{TIM_CLK}{(PSC+1)(PRD+1)}$, where PSC is the prescaler value and PRD the period value.
- The counter period is the maximal value of the counter, i.e. at the next cycle, the counter value will reset to 0.
- The pulse parameter contains the value to compare to the timer counter.
- Mode contains the action to perform on channel 2.

Now that you have everything set up in the `.ioc` file, you can regenerate your code by saving the `.ioc`. The two functions that you will need to use the timer are:

- `HAL_TIM_OC_Start(&htim4, TIM_CHANNEL_2);` to start the timer and the blinking.

- `HAL_TIM_OC_Stop(&htim4, TIM_CHANNEL_2);` to stop the timer and the blinking.

You can now write a simple code that takes advantage of both the interrupt for the button and the timer. Therefore, your new code (not the one generated by CubeMX) should not contain any call to `HAL_GPIO_...` functions, as the LED pin is not in GPIO mode anymore.

! main infinite loop

In bare-metal embedded programming, the `main` function must never return, otherwise the processor might to execute random pieces of memory. Therefore, the `main` must always contain an infinite loop. This loop may be as simple as `while (1) {}`, if there is nothing to do.

1.5 Introduction to power consumption monitoring

Now that you don't have any code left in the main loop, what can we do with it? As we are developing an embedded solution, we have an energy budget determined by the capacity of a battery or by the energy that we can harvest from the environment (or both of them). Therefore, we want to find a solution which has the lowest power consumption to extend the lifetime of our device and/or having our solution functional most of the time.

An easy way to measure instantaneous power consumption is using an ammeter. You simply need to break the power supply line of the MCU and insert the ammeter. Hopefully, this is made easy with a removable jumper placed on the Nucleo MCU board. Find the jumper **JP5** on the Nucleo MCU board, which is located near the LED that is used for blinking as shown in **Figure 7**. Remove the jumper and **DO NOT LOSE IT!** Set up the multimeter in ammeter mode in the mA range and observe the values.



Fig. 7: Remove JP5 to be able to measure the current drawn by the MCU.

i Good practice regarding measurement instruments

It is always a good practice to test your measurement instruments on a reference signal before starting measuring what you need. It might sounds obvious, but in practice it can happen that your probe or cable is broken, a blown fuse, ...

An other method to measure the current of the MCU is using a series resistor called shunt resistor as in **Figure 8**. You need to probe the two sides of the resistor to calculate its voltage drop and then apply Ohm's law to find the current. When measuring with an oscilloscope, the sampling rate of the scope is much faster than the one of the multimeter, therefore it is better to use this method in case of rapid, high consumption current peaks, such as a wireless packet transfer. Practically, you can replace the jumper by a resistor of your chosen value on the pins of **JP5**. You can **NOT** probe the voltage drop with one probe because the ground pin can **MUST** be connected to ground.

on avait calculé : 600 Ω

on a pris : 150 Ω pour pas que la R fasse que ça consomme + que normal

9 et pas trop petite pour qd min pouvoir mesurer une \pm de tension

boucle vide :

0,64 V et 0,66 allumé

→ current = 4.27mA

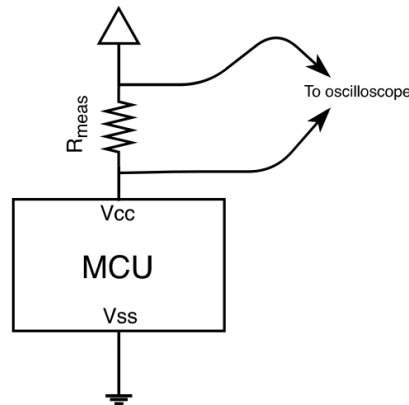


Fig. 8: How to measure the current of an MCU.

⚠ Choosing the right shunt resistor value

The resistor that you choose has a huge impact on your measurements, having a too small voltage drop compared to the noise floor of your oscilloscope will lead to wrong measurements, but having a too high voltage drop will lead to a Brown-Out-Reset, where the MCU resets itself because there is not enough voltage. In the case of the STM, the minimal VDD is 1.71V.

? Comparing empty while loops

We want you to compare the power consumption of 3 different ways of doing nothing in the main loop:

- The first one is already implemented in your code, it is having an empty while(1) loop. *0.64V*
- Now, add a HAL_Delay function in the loop of arbitrary delay. *0.79V → consomme car il doit compter le temps s'écouler*
- Instead of the delay function, put the MCU to sleep mode by calling `__WFI()` which stands for *wait for interrupt*. *0.36V*

This is just a really basic introduction to the power modes management: the goal here is to realize the strong link between your code and the power consumption of your solution. Later in the project, you will learn more about these considerations and implement mode advanced power saving modes.

1.6 (OPTIONAL) To go further: Debugging your code

If you already got involved in some kind of programming, you know that it never goes as expected at the first shot (at least, not often). You therefore need to start debugging your code to understand where things go wrong and fix the issue. Of course, you can use `printf`'s to debug your code. However, as it has been said before, the ST-Link is also a debugger. We can halt the MCU at a specific line of code, look at the values of variables and registers, advance in the code step by step, and many others functionalities. To access the debugger mode, you need to launch your code with the "green bug" button just on the left of run button you use for normal flashing. You can switch to the debugger perspective as proposed by the IDE. Next, we detail how to use the debugger functionalities :

- **Breakpoints** : are lines or code that trigger a halt on the MCU. They allow to stop the MCU at a specific line of code, and observe its state. To set up a breakpoint, simply double click on the

line number where you want to put your breakpoint. The MCU will halt before the execution of this line. You should see a small blue point at the left of the line number as in line 93 of [Figure 9](#)

```

88  /* Initialize all configured peripherals */
89  MX_GPIO_Init();
90  MX_LPUART1_UART_Init();
91  /* USER CODE BEGIN 2 */
92  RetargetInit(&hlpuart1);
93  printf("Hello world!\r\n");
94

```

Fig. 9: Breakpoints in the IDE

- **Step by step execution** : of the code is performed thanks to the buttons in the toolbar as in [Figure 10](#). The play and pause buttons allow starting and stopping the MCU, and the 3 rightmost buttons allow advancing step by step in the code.



Fig. 10: Debugging toolbar

- **Register values** : can be observed in the right panel. The **register** tab contains the values of the registers of the ARM Cortex and the **SFR** (Special Function Register) tab as shown in [Figure 11](#) contains the values of the peripheral registers.

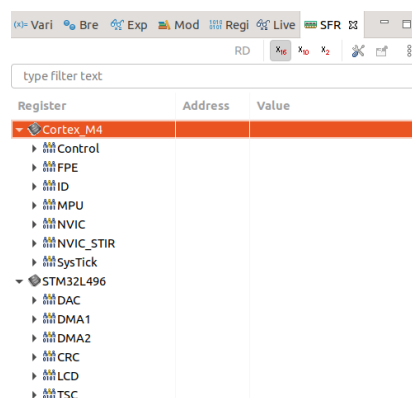


Fig. 11: Special function register values

? Getting used to the debugger

- Set up a breakpoint at the line of the "Hello World!" printf. Launch the debugger and observe that it does not print anything in the serial console.
- Now, advance step by step with your code until you reach the line where you start your timer. Take a look at the registers of TIM3 and observe the changes. (You can compare what you observed with the reference manual of the MCU.)

Finally, when you debug real-time software, don't forget that, when reaching a breakpoint, your code is not real-time anymore ! A few printf can also be helpful...

! Don't abuse of printf

Be aware that the printf function is hungry in CPU cycles. If you decide to print too much information and use a slower clock speed on your MCU, you might miss real-time constraints.

2 Demo D2a: power consumption

During this LELEC2102 project, we expect you to be able to present quick demos on specific topics, as scheduled in the program for this semester. They can be seen as small checkpoints to make sure you master the important basic blocks of the project. These demos should take you less time than writing a detailed report while still providing a good occasion to develop new skills, learn by doing and finally get some feedback along the way! There is no need to write anything for a demo but you must make sure it will run "live" smoothly.

The **demo D2a** will focus on the power consumption of your MCU. Indeed, you programmed it and learned how to implement some ideas in practice: it is now time to see how much power it draws and if it works as expected. It is critical for embedded programming as you will need to fit into a power budget. For this demo, we expect you to:

1. Show what is the impact of a LED blinking on the power consumption trace. Is it significant for embedded computing applications? Therefore, what is a good practice for battery-powered nodes?
2. Observe the power consumption with and without WFI(). What is the difference? Why is it so?
3. Observe how the power consumption is changing if we change the clock frequency. What could cause this important change?


2) WFI

P simple dans la R sa
consommation max de la LED : 0,05V
R = 150Ω

$$P = \frac{0,05^2}{R} = 16 \mu W$$

$$P_{\text{moy}} = \frac{P}{2} = 8 \mu W$$

AP qd LED clignote est > en WFI que en ¹² delay car en delay il consomme + donc LED a moins de puissance


$$I = \frac{V}{R}$$

$$P = I \cdot V$$

vu que tension fixe ($V = 3.3V$) on peut parler en mA au lieu de W

on divise freq par 2 : avec bande vide : $0.64V \rightarrow 0.43V$