

# SERIAL COMMUNICATION BUSES

Lab materials



## Contact

**Dimitri Boudier** – In charge of this course

[dimitri.boudier@ensicaen.fr](mailto:dimitri.boudier@ensicaen.fr)

## Resources

All the course materials (lecture slides, lab, hardware and software tools, ...) are on the Moodle page of the course:

<https://foad.ensicaen.fr/course/view.php?id=213>

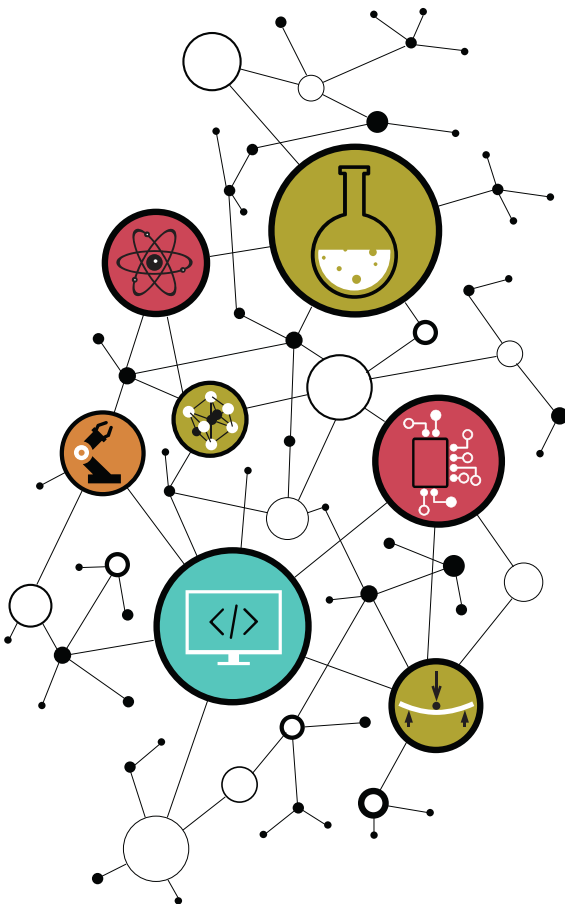


Except where otherwise noted, this work is licensed under <https://creativecommons.org/licenses/by-nc-sa/4.0/>

Contact.....	2
Resources.....	2
<b>Part 1 Prelude.....</b>	<b>5</b>
I. Set the context.....	6
I.1. Sensors and systems connectivity.....	6
I.2. How to work.....	7
II. Hardware and software needs.....	8
II.1. In short.....	8
II.2. Hardware and software presentation.....	9
III. Starting with the environment.....	12
III.1. Creating a project.....	12
III.2. Generated project insight.....	13
III.3. Control the GPIOs with the HAL functions.....	14
III.4. Use interrupts.....	16
<b>Part 2 Asynchronous serial communication / UART.....</b>	<b>21</b>
I. Presentation.....	22
II. Specifications.....	23
III. Implementation on a STM32.....	24
III.1. Project creation.....	24
III.2. Generated project insight.....	25
III.3. UART transmission.....	26
III.4. UART reception.....	29
III.5. Bridge UART (FRENCH ONLY).....	32
IV. Overview.....	35
<b>Part 3 SPI – Serial Peripheral Interface.....</b>	<b>37</b>
I. Specifications.....	38
II. Frame constitution.....	39
III. Application: Light click.....	40
III.1. Daughterboard study.....	40
III.2. SPI peripheral configuration.....	41
III.3. light_click driver.....	42
IV. Application: Accel 2 click.....	47
IV.1. Sensor study.....	47
IV.2. SPI peripheral configuration.....	48
IV.3. accel_2_click driver.....	49
V. Overview.....	57
<b>Part 4 I<sup>2</sup>C – Inter-Integrated Circuit.....</b>	<b>59</b>
I. Specification.....	60
II. Application: Weather click.....	61
II.1. Sensor study.....	61
II.2. I <sup>2</sup> C peripheral configuration.....	62
II.3. weather_click driver.....	62
II.4. Overview.....	73
<b>Part 5 Glossary.....</b>	<b>75</b>



PART 1  
**PRELUDE**



### I. Set the context

This document and all of the surrounding materials have been built for the “Sensors and Systems Connectivity” course, for the third-year students in GPSE-IPC.

However this document can also be used outside of this course, especially for those who want to go deeper in their understanding of embedded systems. The required hardware and software are listed in the next parts (II. Hardware and software needs page 8), with some of the hardware equipment being available for lend or directly accessible in the A203 classroom.

#### I.1. Sensors and systems connectivity

This document is the first step of the “Sensors and systems connectivity” course. The aim of the latter is to make an introduction to communicating sensors at first, followed by an introduction to communicating systems.

Here the “communication sensor” term refers to any measuring element that is capable of exchanging information with a not-so-distant processor. Modern systems use a digital interface, even if some analogue interfaces still live on. Those digital exchange interfaces are called **buses (or communication buses)** and their use is widespread (if not always used) in the embedded systems ecosystem.

As you could have guessed, a communication bus is a communication system that shares common links between various system’s components. In an embedded system, the communication bus (or the buses) are usually implemented onto the **PCB (Printed Circuit Board)**, or on few side PCBs that are attached to the system’s main PCB. The system’s main processor (generally a **MCU, MicroController Unit**) is called the master (or controller) and it drives the communication bus. It communicates with other devices (sensors, actuators, other MCUs, ...) that are called slaves (or targets).

This course will deal with three communication buses: **UART, SPI** and **I<sup>2</sup>C**. They are by far the most encountered in embedded systems. To work with these buses, we will make an **MCU** communicate with various **sensors**, by developing the corresponding **drivers**. It will be necessary to study the technical documentation of those integrated circuits, and then to analyze and decode the data frames with an logic analyzer.

### 1.2. How to work

All of the files provided in this lab archive are meant to be enough to follow this course on your own. You will find in this archive the following directories:

- **datasheets** manufacturers technical documentations
- **lectures** lecture materials for deeper insight about the communication buses
- **tutorials** how-to documents for hardware and software
- **workspace** the workspace (directory) into which the STM32 projects will be created
  - **drivers** drafts of the sensors' drivers (to be completed by yourself)

This document aims with studying serial communications by using three different sensors (light, acceleration, temperature+humidity+pressure). If want to follow this lab on your spare time, feel free to ask for the sensors to the teacher.

We can also provide you with other sensors (list of available sensors in the **datasheets/** folder) if you want to explore even more and get a more solid experience in driver's development. Note that developing a driver from scratch is very educational, since you have to be able to extract useful information out of the datasheet, that can be big and sometimes complex.

#### 1.2.a. Assessment

For those who follow the « Sensors and Systems Connectivity » teaching, one mark will be given to the report that you will hand to the teacher at the end of the course

For that you have to answer the ✍ questions, provide the 🖼 screenshots (serial terminal, logic analyzer), and complete the 📄 source files.

You can answer on this digital document (the PDF and editable versions are both in the archive), or on a paper document (the one printed for you, or another one made by yourself). The only restraint is that the questions must be fully written to ensure no question has been forgotten). Screenshots should ideally be directly integrated into the document (paper or digital).

Source files will be requested on the Moodle page (do not forget to fill the **@author** fields on the beginning of each file). The source files will be requested as we go along.

#### 1.2.b. Notation

- ✍ Question waiting for a written answer (by analyzing all given resources)
- 📄 Answer by writing code, programming, debugging, ...
- 🖼 Answer with a screenshot (serial terminal, oscilloscope, logic analyzer, ...)

## II. Hardware and software needs

### II.1. In short

Here is a short list of our hardware and software needs, all of those being used in this document. A detailed presentation is provided on the next pages.

The hardware equipment used in this document is:

- a NUCLEO-L073RZ development board
  - any other NUCLEO board (-L746RG, -WL55JC1, ...) is compatible.
- an Arduino UNO click shield;
- three MIKROE Click Board:
  - In this document: USB UART click ; Light click ; Accel 2 click ; Weather click
  - In the classrooms: Air Quality click, Pollution click, Barometer click, ...
- an IKALogic logic analyzer:
  - Any other logic analyzer or oscilloscope is compatible, as long as they support protocol decoding (UART, SPI and I<sup>2</sup>C)
  - *Note: this is the only optional equipment of this list (but still very useful)*

The software are the following:

- STM32CubeIDE (validated with version 1.11.0) :
  - IDE used for programming and debugging the STM32 NUCLEO boards
  - <https://www.st.com/en/development-tools/stm32cubeide.html>
  - any other compatible IDE (Keil, VS Code with the STM32 extension, ...) is accepted
  - *Note: there is a tutorial in this archive for STM32CubeIDE (but not for the others IDEs)*
- ScanaStudio
  - Software dedicated to the IKALogic logic analyzer
  - <https://ikalogic.com/scanastudio/>
  - If you use another logic analyzer or oscilloscope, you must use the associated software
  - *Note: there is a tutorial in this archive for IKALogic and ScanaStudio*
  - *Note: it this the only optional software of this list*
- A serial terminal
  - Any should do the trick
  - Tera Term (Windows), PuTTY (Windows/Linux), GTKTerm (Linux), minicom (Linux), ...
  - *Note: there is a tutorial in this archive for the four listed terminals*

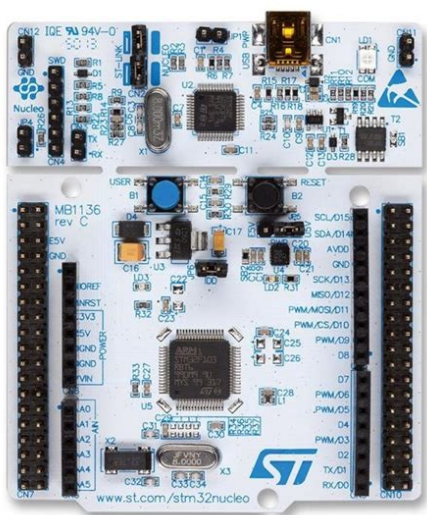
Remember that there are how-to documents in this [tutorials/](#) folder of this lab archive. They present the hardware and/or software and show how to use them. Keep visiting this folder anytime you encounter a new equipment.



## II.2. Hardware and software presentation

### II.2.a. STMicroelectronics NUCLEO boards

The French-Italian firm STMicroelectronics is one of the world leaders on the semi-conductor markets and more precisely on the market of 32-bit ARM-based MCUs. In order to help developers to take their MCUs in hand, ST offers various evaluation boards. Their most well-known evaluation board is the **NUCLEO**, being cheap while have a high diversity of MCUs (thus matching with a wide range of uses and application, from low-power to high-performance MCUs).



As an example, the **NUCLEO-L073RZ<sup>1</sup>** board used in this document has on its PCB the target MCU (STM32L073RZ), a programming/debugging probe, a push-button and a LED.

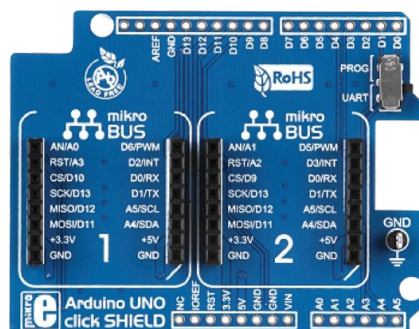
The NUCLEO boards also come with two connectors that give an easy access to the MCU pins. This facilitates the prototyping process. You can see on this board an Arduino connector and a Morpho connector, the latter being specific to STM (2 x 38 pins).

Last but not least, the NUCLEO boards are also equipped with a serial communication interface through the USB port. This functionality is called STLink VCP (*Virtual Comm Port*). This offers a user-friendly debugging interface, with no additional equipment needed.

For more information please see the dedicated tutorial document: [tutorials/tutorial1\\_nucleol073rz](https://www.st.com/en/evaluation-tools/nucleo-l073rz.html).

The processor embedded onto the NUCLEO-L073RZ board is a STMicroelectronics **STM32L073RZ<sup>2</sup>** MicroController Unit. It is a 32-bit MCU built around an ARM Cortex-M0+ CPU. Its main characteristic is being an ultra-low power MCU (the 'l' in "STM32L" stand for "Low-power"). Among its numerous peripherals we can name the USART, SPI and I<sup>2</sup>C: these are the ones that you will work with during this lab.

As said above, all NUCLEO boards come with an Arduino connector. We will plug a **MIKROE Arduino UNO click shield** onto the NUCLEO board. This will give us two Click board slots into which we will place the Click boards used in this lab. The Click boards contain the sensors, as explained on the next page.



1 <https://www.st.com/en/evaluation-tools/nucleo-l073rz.html>

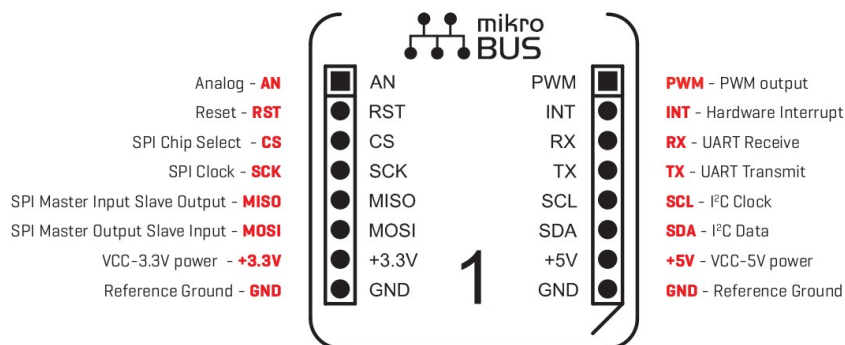
2 <https://www.st.com/en/microcontrollers-microprocessors/stm32l073rz.html>

### II.2.b. MIKROE Click board™

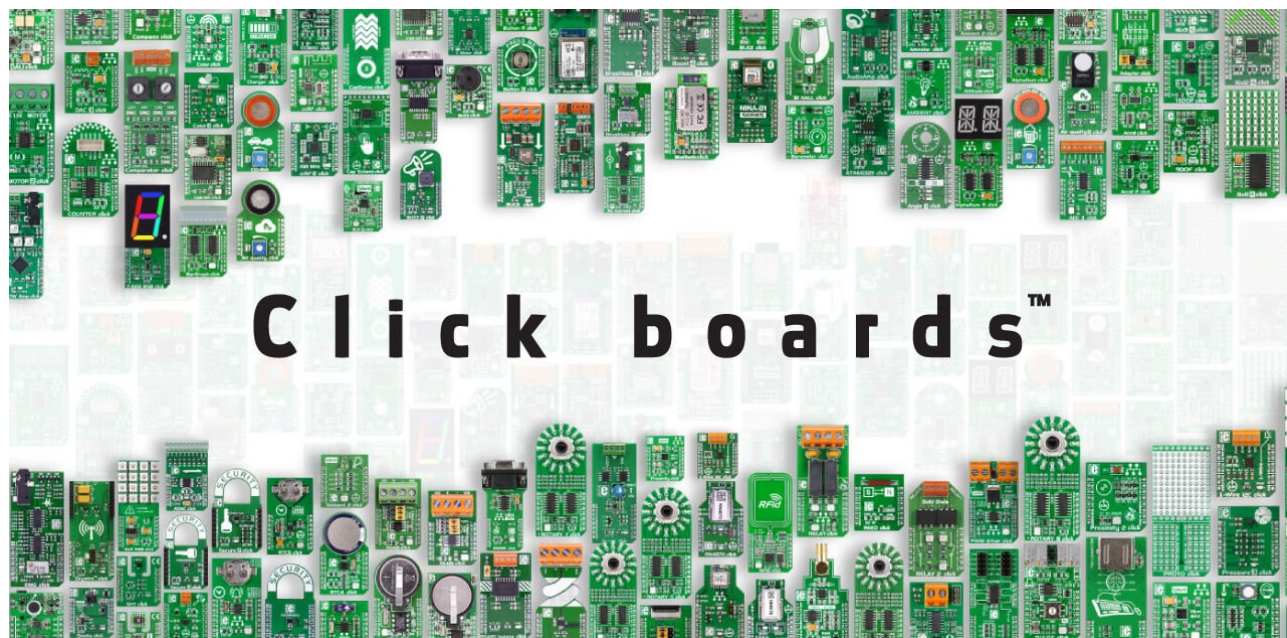


MikroElektronika<sup>3</sup> (shorten in MIKROE) is a Serbian company specialized in hardware and software conception.

The firm has developed the open mikroBUS™ standard, which defines a connector format in which all of the pins are fixed. The choice of the signals that are integrated into this format (GND, +3.3V, +5V, UART, SPI, I<sup>2</sup>C, analogue, PWM) makes the mikroBUS™ compatible with a wide set of different uses.



The mikroBUS™ standard helped MIKROE to spread its lead product: the Click board™ series. Each of those boards possesses a unique device (sensor, actuator, power, communication, ...), but they all share the same mikroBUS™ format. As of September 2023, more than 1,500 Click boards are active. Many evaluation boards are now equipped with mikroBUS™ slots in order to host Click boards (just like the Microchip Curiosity HPC used in first year of ENSICAEN).



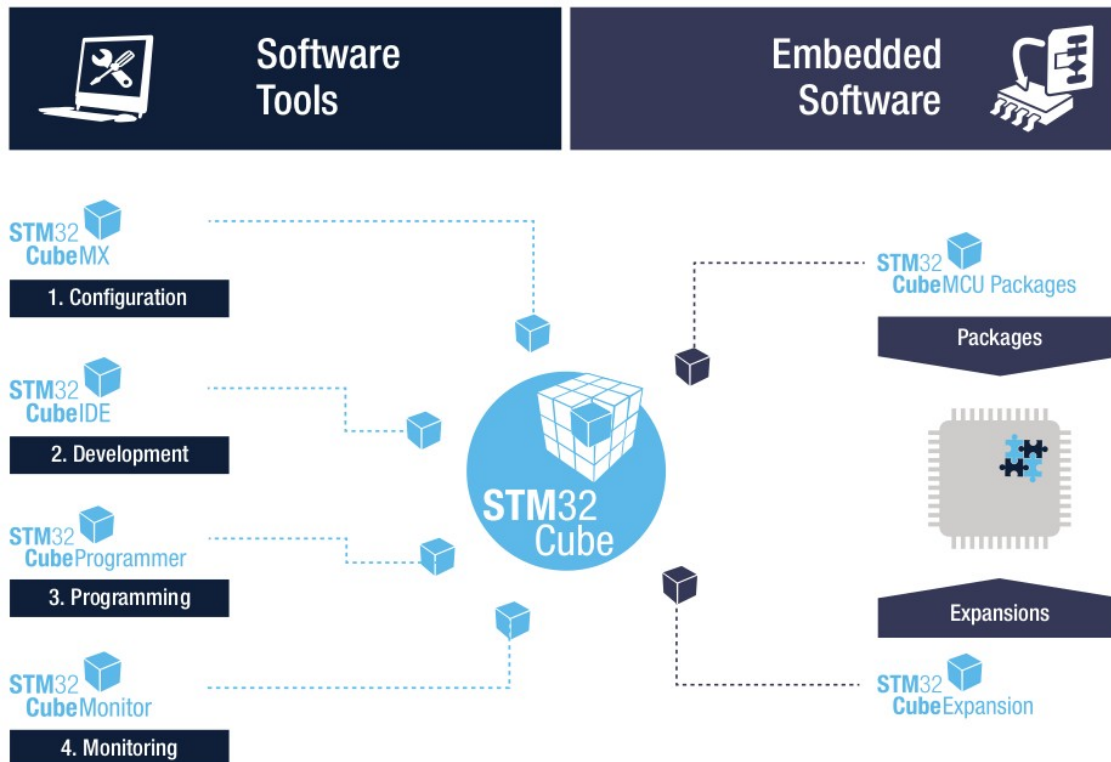
For each Click board, MIKROE gives access to the board schematic, to the ICs' technical documentation and to the drivers they have developed. Everything is available on the manufacturer website. For the sensors used in this lab, all the necessary files have been gathered in the [connectivity/datasheets/](#) folder.

3 <https://www.mikroe.com/about>

### II.2.c. IDE – Integrated Development Environment

For our firmware developments on STM32 MCUs we will use the software suite given by STMicroelectronics: **STM32Cube**<sup>4</sup>. It is an entire ecosystem that contains many software tools, though we will focus on only two of them:

- **STM32CubeIDE**, the IDE;
- **STM32CubeMX**, an initialization code generator.



The **Integrated Development Environment** (that will be called **IDE** from now on) is called **STM32CubeIDE**. It uses the Eclipse framework, which is the largest cross-platform open-free IDE. Eclipse works with perspectives, i.e. sets of windows that are configured for specific phases of the development (e.g. edit, debug).

We could have decided to program the MCU at a register-level, just like the Embedded Systems lab in first year. But we would have needed much more time for this course because ARM Cortex-M CPUs are more complex as compared to PIC18 MCUs.

As we wanted to focus on the sensors' drivers rather than on the CPU/MCU, we will use **STM32CubeMX**. It is a tool with a graphical interface that lets us choose the configuration of the MCU and its internal peripherals. A fully functional firmware can be built in few minutes, even with barely no knowledge of the device. In a professional context this tool is used to accelerate the prototyping phase and thus reduce the Time-to-market of software embedded solutions. We must confess that this kind of tool also exists for Microchip, even if we hid it to you (for teaching purpose obviously). This one is called MCC (MPLAB Code Configurator).

STM32Cube is cross-platform, meaning that you can work either on Windows or Linux. For downloading, setup and use instructions, please see the [tutorial\\_stm32cubeide.pdf](#) file.

4 <https://www.st.com/en/ecosystems/stm32cube.html>



### III. Starting with the environment

Before going deep into the study of serial communication buses, we will take some time to discover our work environment by focusing only on the NUCLEO board and the STM32CubeIDE software.

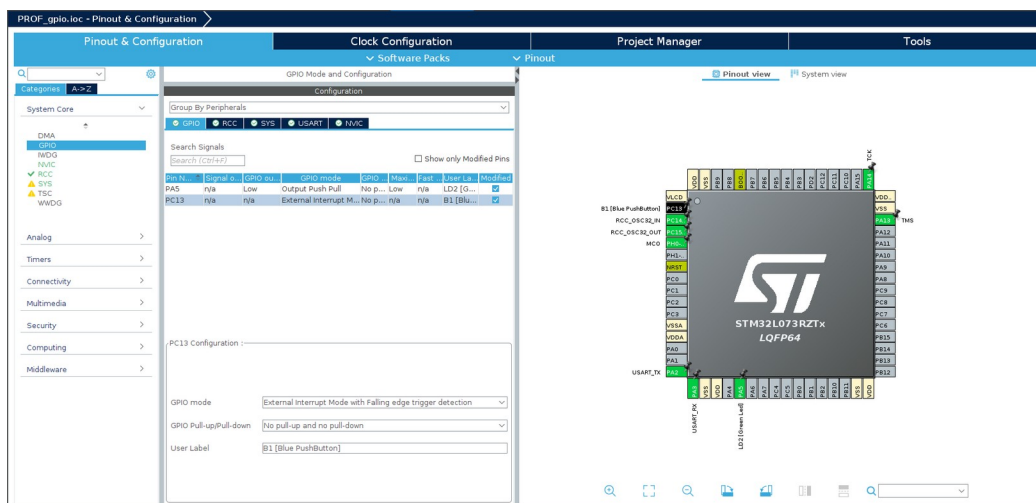
#### III.1. Creating a project

📖 Read the `tutorial_stm32cubeide.pdf` file to create a project with the STM32CubeIDE software. **Be careful** to consider the information below while following the tutorial's steps.


📖 You will need the following information while **creating** the project:

- **Board selector**: use the NUCLEO board reference (bottom side of the board);
- **Project Name**: YOURNAME\_gpio ;
- **Project Location**: select the `connectivity/workspace/gpio/` directory, in your lab archive;
- **Initialize all peripherals with their default Mode ?**: Yes.

After the creation phase, a window dedicated to the configuration phase will open. Let us have a look at the peripherals that are configured by default.



🔍 In **System Core** → **GPIO**, note here the pin number of the configured GPIOs, and their function as well:

📖 The configuration has been prepared by STM32CubeMX (with this window), but the code generation is still to be done: click on **Project** → **Generate Code** (or the  icon).

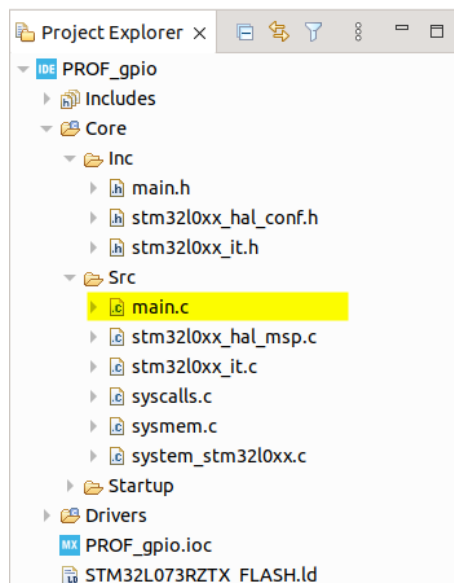
Et voilà ! Two GPIOs are ready to use: the one attached to the LED and the one attached to the push button. Their initialization code has been written, alongside with manipulation functions and there are yours to use from now on. We will see them on the next page.

### III.2. Generated project insight

The tab at the left-side of the IDE is the *Project Explorer*.

At first the tree view of the project seems to be complex, but it is actually quite easy to understand:

- **Core/Src/**: directory that contains the source files (C and asm) **for the application** (thus matching specific requirements);
- **Core/Inc/**: same, but for header files (.h)
- **Drivers/STM32L0xx\_HAL\_Driver/Src/**, **Inc/**: folders that respectively contain the source and header files for the HAL (Hardware Abstraction Layer). In other words, these are the driver functions (or BSP) given by CubeMX for the peripherals manipulation. Those files are **specific to the target STM32 MCU**.
- an **.ioc** file that opens the configuration window (CubeMX), as seen before.



A deeper insight is provided in the [tutorial\\_stm32cubeide.pdf](#) file.

📖 Open the `Core\Src\main.c` file. You should notice that it is already partially completed. This is the consequence of using CubeMX (an initialization code generator) after the project creation.

📖 Browse the `main()` function. You should see that some initialization functions are called, followed by an empty `while(1)` loop.

📖 Hold the `Ctrl` key and `click` on the `MX_GPIO_Init()` function. This `ctrl+click` shortcut brings you to the function definition, which in this case is in the `main.c` file. It has been created whilst you were in the project configuration phase. By the way you can see that the function's instructions match the parameters that you have seen in the graphical configuration interface. Right after these instructions, the `HAL_GPIO_Init()` HAL function is called.

📖 `Ctrl+Click` on the `HAL_GPIO_Init()` function call. This time another file opens: `stm32l0xx_hal_gpio.c`. This file belongs to the HAL (Hardware Abstraction Layer) and contains all the configuration and manipulation functions for the GPIO peripheral. Among these functions, you will use:

- `HAL_GPIO_Init()`: function that configures the pin as a GPIO;
- `HAL_GPIO_ReadPin()`: function that reads the logic level of the specified pin;
- `HAL_GPIO_WritePin()`: function that sets the logic level of the specified pin.

To keep it short the HAL (*Hardware Abstraction Layer*) is a set of functions written for setting up and controlling the MCU peripherals. They are provided by STMicroelectronics. For a detailed presentation, please read the [tutorial\\_stm32cubeide.pdf](#) document, from the `tutorials/` folder in the lab archive.

### III.3. Control the GPIOs with the HAL functions

The NUCLEO-64 boards (family to which the NUCLEO-LR073 belongs) are equipped with a push button and a LED. You have already check (in part III.1 Creating a project) that those GPIOs have been configured and are now ready to use with the functions previously listed.


#### !/\ WARNING /\!

The good news when using STM32CubeMX is that you do not have to write the content of the `main()` function. Indeed the initialization code generator takes care of it, sparing your time.

Obviously the written code does not fit with every single application, meaning you are free to add your own code. But you must respect the places that CubeMX gives. These places are marked out by two matching comment tags (e.g. `/* USER CODE BEGIN WHILE */` and `/* USER CODE END WHILE */`).

**As a consequence, the code that you will develop shall be written between two comment tags `BEGIN` and `END`.**

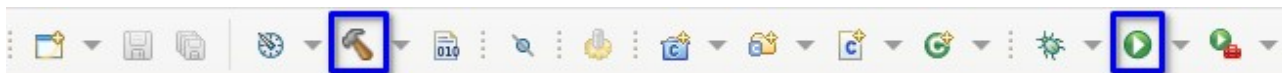
Any code line outside of the marks will be deleted when regenerating the code with CubeMX (for instance when adding support of another peripheral, which we will do later on). On the other hand, any code line between two matching tags will remain. It is a very simple rule yet a strict one.

 In the `while(1){...}` loop (`main()` function), write the following code

Beware: write this code between two matching comment tags.

```
if( HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET )
{
    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
}
else
{
    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
}
```

 Compile (Project → Build All) and launch the program onto the target MCU (Run → Run).



You will quickly guess how to interact with the NUCLEO board to confirm that the program runs perfectly.

📖 Ctrl+click on `LD2_GPIO_Port`. You should observe that several constants are defined at the very same place.

🔪 Which file contains these constants definitions? Who/what defined them here?

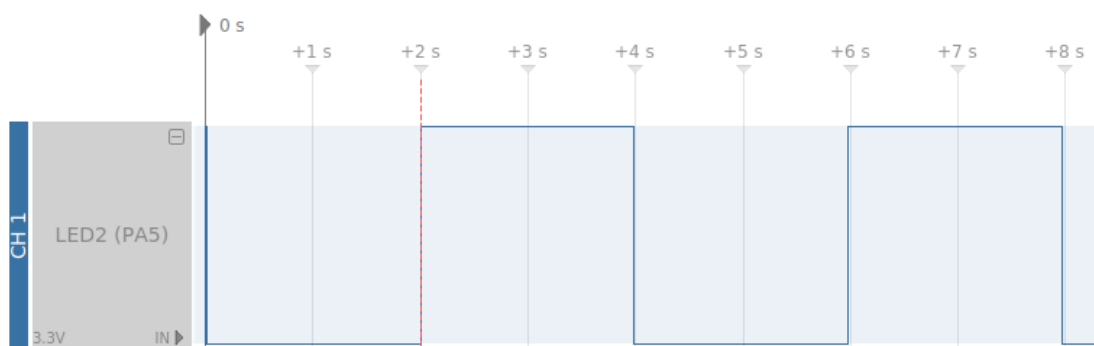
🔪 Which function let us change the GPIO logic state? Take a look among the numerous functions declared in the `stm32l0xx_hal_gpio.h` file.

🔪 Which function let us set a software delay (thus being a blocking function)? Look into the `stm32l0xx_hal.h` file.

📖 Thanks to both functions, make the LED blink by changing its logic state every two seconds.

📖 Confirm visually.

📖 Confirm the blink period with an oscilloscope or a logic analyzer.

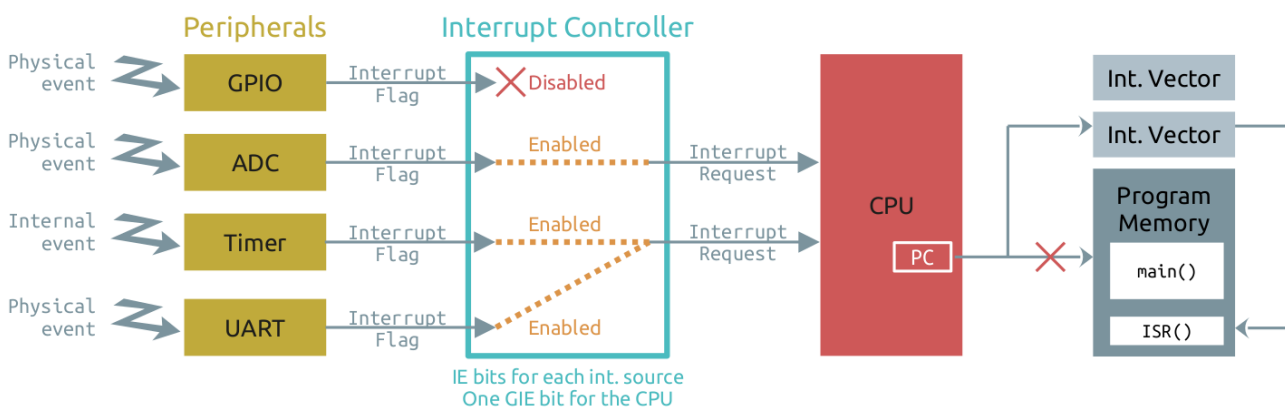


*Example of what you should observe with a logic analyzer.*

In just few minutes you took control of a rather development board that contains a rather complex STM32 MCU. You configured two GPIOs and controlled them as input and output. The use of the IDE (more precisely of STM32CubeMX) was very helpful in term of work time.

### III.4. Use interrupts

As any self-respecting micro-controller unit, the STM32 MCUs can use an interrupt mechanism. As a reminder the goal of the interrupts is to let the peripheral “catch” an event, in which case it rises an **IF (Interrupt Flag)** bit. If one event is allowed to interrupt the running program, the Interrupt Flag becomes an **IRQ (Interrupt Request)**, which is an electric signal linked to the CPU. When one IRQ gets to the CPU, the latter pauses the running program and then goes and executes another code portion stored in a memory zone called **Interrupt Vector**. This vector usually contains few instructions, e.g. a call to the function that is actually in charge of dealing with the incoming event: the **ISR (Interrupt Service Routine)**.



For ARM cores (such as the one in the STM32 MCUs), the interrupt controller is called **NVIC (Nested Vector Interrupt Controller)**. The GPIOs that are configured as inputs can trigger an interrupt following a change in their logic state. This can be set with the **EXTI (External Interrupts) Controller**, a sub-stage of the NVIC. In this section we will configure the push button input so that it triggers an interrupts whenever the button is pressed down.

Open the `VOTRENOM_gpio.ioc` file. It will open the MCU graphical configuration interface.

📖 On the « *Pinout view* », click on the push button pin.


🔪 In which mode is it configured? Note that this is *not* `GPIO_Input`.

📖 On the left tab: *System Core* → *GPIO*, click on this pin in the table.

🔪 In which « *GPIO mode* » is it configured? What does it mean?

📖 In *System Core* → *NVIC*, check the line `EXTI line 4 to 15 interrupts` and set the value of the `Preemption Priority` to '1'. In ARM CPUs, '0' is the highest priority level. We want our push button to be of a lower priority than the `HAL_Delay()` (which has a level priority of '0').



Click on **Project** → **Generate Code** (or ) to generate new code.

The STM32CubeMX configurator has updated the existing code, by adding the changes that you just have requested.

### **/!\ Reminder /!\**


In order to modify the existing code, STM32CubeMX finds its way thanks to the comment tags that we discussed about (e.g. `/* USER CODE BEGIN Init */` and `/* USER CODE BEGIN Init */`).

Recall that you must write your own code between two matching tags, otherwise anything will be deleted anytime you ask for the code to be generated again.

✎ In the `MX_GPIO_Init()` function (`main.c`), which instructions have been added? What are they used for?


In the `stm32l0xx_hal_gpio.c` file the `HAL_GPIO_EXTI_IRQHandler()` function is the Interrupt Service Routine (ISR), called after an Interrupt Request (IRQ), which is triggered by a logic state change on a GPIO input. The ISR then calls another function: `HAL_GPIO_EXTI_Callback()`. This function is the one given to the developer (you) to deal with the event.

✎ This callback function is declared few lines further in the same file. What does the `__weak` keyword means in its declaration: `__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)?`

 In the `main.c` file, copy and paste these lines (note that the comment tags are given).


```
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == B1_Pin) {
        for( int i = 0 ; i < 10 ; i++ ) {
            HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
            HAL_Delay(50);
        }
    }
}
/* USER CODE END 0 */
```

You should notice that the function prototype is exactly the same as the one of the previous question: it is a function redefinition (made possible thanks to the `__weak` qualifier). It is up to the developer to fill the definition of this function.

 What does this function do?

 Upload the program in the target MCU and verify its behavior.

You should observe that the previous application (blinking every two seconds) is still running. But now it is interrupted by another code portion whenever the push button is pressed down.

 Confirm with an oscilloscope or a logic analyzer, triggered on the falling edge of the push button pin.

### The end

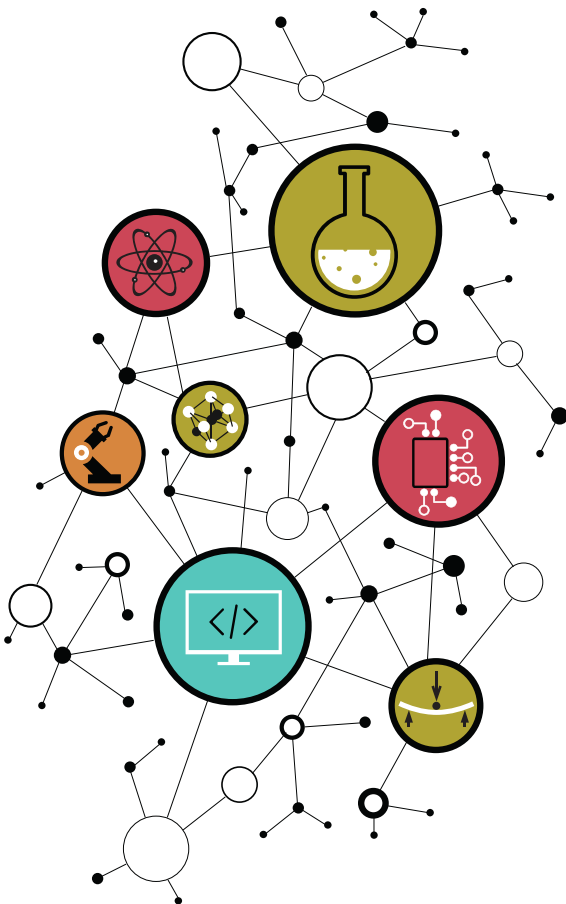
The interrupt mechanism, which is a rather complex mechanism in MCUs, has been set and seen in few minutes. On one hand the automatic configuration did not really help to have a better understanding of the mechanism, but on the other hand the abstraction layer given by CubeMX kept us from spending a lot of time on the application development.





PART 2

# ASYNCHRONOUS SERIAL COMMUNICATION / UART

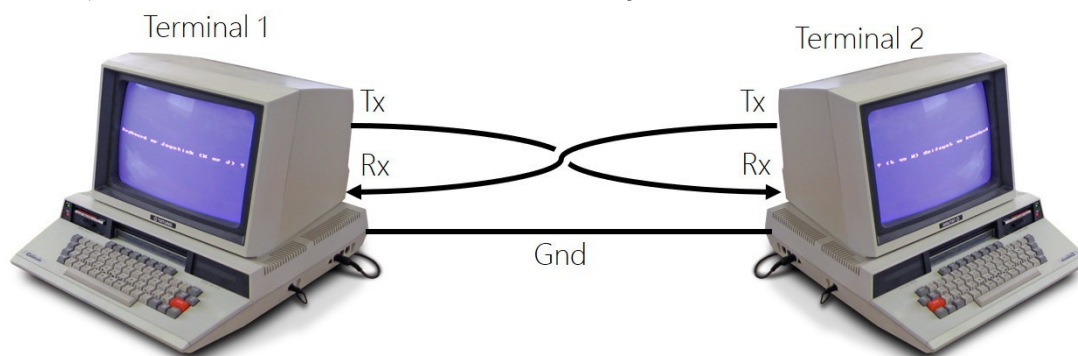


### I. Presentation

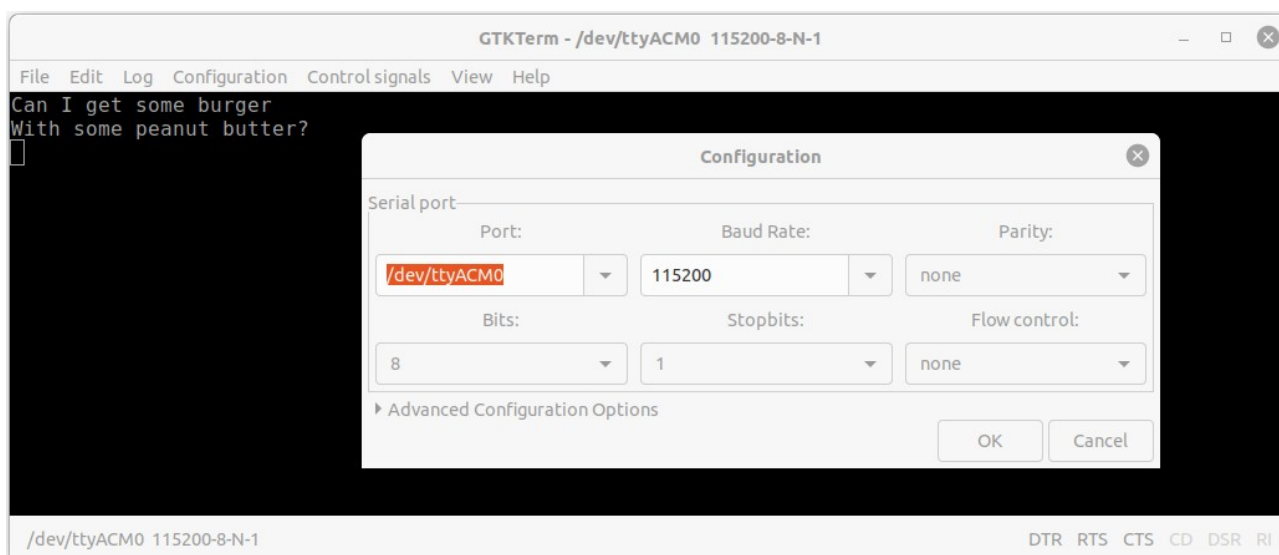
The **UART** is an **Universal Asynchronous Receiver-Transmitter**. It names a device or a peripheral but it is not a communication protocol, whatever the misuse of language frequently seen. The communication between two UARTs is an **asynchronous serial communication**.

This communication has disappeared from the computer world (the USB arrived in 1996 and replaced it), but it still remains in the embedded world. Indeed this peripheral is easy to build and to use, and allows anyone to quickly set up a communication interface.

The minimalist UART device is made with two lines: a transmission line called **Tx** and a reception line called **Rx**. To connect two UARTs together, both lines should be crossed ( $Tx_1 \rightarrow Rx_2$  and  $Rx_1 \leftarrow Tx_2$ ). The transmission mode is then a **full-duplex**.



Its topology differs from other protocols that are usually met in embedded systems: this is a **point-to-point communication**, and not a proper communication bus. In practice this serial communication is rather used to connect a computer to an electronic system. In this way the computer uses a serial terminal that communicates with the embedded system, giving access to a debugging and/or configuration interface. This proves the simplicity advantage, because a simple software terminal (Tera Term, PuTTY, GTKTerm, ...) can exchange data with the target system, simply by sending ASCII characters.



## II. Specifications

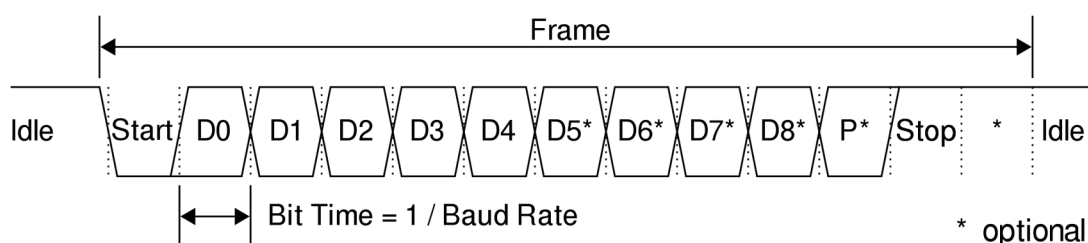
From a hardware view, a UART peripheral contains two lines:

- 1 **Tx** line for the transmission (connected to its counterpart Rx line);
- 1 **Rx** line for the reception (connected to its counterpart Tx line).

The lines idle state is imposed by the serial communication protocol. While no data is exchanged, the lines keep a high logic state. Note that UARTs in embedded systems use TTL voltage levels, i.e. 0 V for a low logic state and +3.3 V/5 V for a high logic state.

The communication protocol also rules the frame structure, even if it is slightly adaptable. A data frame is made of several fields:

- **1 start bit** (low logic state, to “break” the idle state);
- **5 to 9 data bits (payload)**, LSb<sup>5</sup> first;
- **1 parity bit**, optional, used for error detection;
- **1 or 2 stop bits** (high logic state).



The most widespread configuration is the 115200-8N1 :


- 115200 is the baud rate (usual values range from 9600 to 115200 baud) ;
- 8 data bits (from 5 to 9) ;
- *No parity bit* (N = *No parity bit* – O = *Odd parity bit* – E = *Even parity bit*) ;
- 1 stop bit (1 or 2).

Other asynchronous serial link standards exist, such as the RS-232 and RS-485 standards. It is not useful to describe them here because there are more used in an industrial context but are rarely met in embedded systems. We will just say that they are quite similar to what have be written above, apart from the lowest OSI model layer (Physical layer) that sets the requirements of voltage levels, bit coding, connector mechanical specification, differential pairs, wire lengths, ... ..

**5** In this document we will use the *LSb/MSb* notation for *Least/Most Significant bit*, and the *LSB/MSB* notation for *Least/Most Significant Byte*.

### III. Implementation on a STM32



#### III.1. Project creation


 Following the `tutorial_stm32cubeide.pdf` file, create an STM32CubeIDE project. Also consider the information below.

 You will need these information while **creating** the project:

- **Board selector**: use the NUCLEO board reference (bottom side of the board);
- **Project Name**: YOURNAME\_uart;
- **Project Location**: select the directory `connectivity/workspace/uart/`;
- **Initialize all peripherals with their default Mode ?**: Yes.

 You will need the information below while the MCU **configuration**:

- Connectivity
  - → USART2
    - → Parameter Settings
      - 115 200 baud ; 8 data bits ; no parity bit ; 1 stop bit
    - → NVIC Settings
      - USART2 Interrupt : Enable
    - → GPIO Settings
      - Note here the MCU pins used by the USART2 peripheral
        -  USART2\_TX = \_\_\_\_\_
        -  USART2\_RX = \_\_\_\_\_

One the configuration is ready: **Project → Generate Code** for generating the code corresponding to the peripherals configuration (or click on the  icon).

Here we go again. Configuration and control functions for the USART2 peripheral are ready in few minutes thanks to the STM32CubeMX initialization code generator.

To give you a rough estimate, the STM32L0x3 *reference manual* contains 1,040 pages and the « USART/UART » chapter contains 67. This USART peripheral uses a dozen registers. Let us compare this to the PIC18F27K40, which datasheets counts 818 pages. Its EUSART peripheral is detailed on 35 pages and uses 6 registers.

For the PIC18 EUSART peripheral, which is about half as much complex as the STM32 USART, you spent about 10 hours developing a small driver for this sole peripheral! Also remember that the Cortex-M0+ is one of the simplest ARM CPUs!



### III.2. Generated project insight

The *Project Explorer* is on the left side of the IDE window.

📄 Open the `main.c` file and browse the `main()` function. There are some initialization function calls, and an empty `while(1)` loop.

📄 Hold the `Ctrl` key and `click` on the `MX_USART2_UART_Init()` function. This brings you to the function definition, in the `main.c` file. Similarly to the GPIOs initialization function, this one has been created when you prepared the USART2 configuration, just after the project creation. You can compare the graphical interface parameters and the instructions of this initialization function: all parameters give birth to C instructions. Right after those parameters, you should see that a HAL function is called: `HAL_UART_Init()`.

📄 `Ctrl+Click` one the `HAL_UART_Init()` function. A new file opens: `stm32l0xx_hal_uart.c`. It is a HAL (Hardware Abstraction Layer) file that contains all the configuration and control functions of the USART peripheral. Among those function, you will use:


- `HAL_UART_Transmit()`: sends data to the Tx line, blocking function;
- `HAL_UART_Transmit_IT()`: same, but non-blocking (uses the interrupt mechanism)
  - Once the transmission is complete, the `HAL_UART_TxCpltCallback()` function is automatically called by the Interrupt Service Routine (ISR). To perform any operation, this callback must be redefined.
- `HAL_UART_Receive()`: reads data from the Rx line, blocking function;
- `HAL_UART_Receive_IT()`: same, but non-blocking (uses the interrupt mechanism)
  - Once a data is received, the `HAL_UART_RxCpltCallback()` function is automatically called by the ISR. TO perform any operation on data reception, this callback must be redefined.


### III.3. UART transmission

With all these functions in hand you will be able to send a message using the UART peripheral in a short amount of time.




#### **! REMINDER !**

You must write your code between two matching comment tags!  
 (e.g. `/* USER CODE BEGIN WHILE */` and `/* USER CODE END WHILE */`)

 In the `while(1)` loop (`main()` function), you will send the string `"Hi\r\n"` through the USART2 peripheral.

 What arguments should you give to the `HAL_UART_Transmit()` function?

 Send this string and add a 2-second delay with the `HAL_Delay()` function, which is defined in the `stm32l0xx_hal.c` file.

 Compile  and upload  into the target. If there is no error, carry on to the next question to confirm the good operation. If there is any error, please solve it or ask for help.

### III.3.a. STLink Virtual COM Port

The NUCLEO board contains the target MCU (STM32WL55JC) but it also has another MCU called the STLink. The latter is an intermediary between the computer and the target MCU, and it mainly operates as a programming and debugging probe.

The STLink MCU also has another functionality: the **STLink Virtual COM PORT (VCP)**. On one side it is able to emulate a serial communication with the computer, through the USB cable. On the other side the USART2 peripheral of the target MCU is physically connected to the STLink. One could say that the STLink acts like a bridge, between the target MCU (UART) and the host computer (USB). This requires no additional device (as compared to the UART/USB FTDI used in 1<sup>st</sup> year) and it appears to be a handy tool, even more for those who will work off session.

However due to its physical connection with the STLink VCP, the USART2 pins are available on the NUCLEO connectors (neither the Morpho (PA2/PA3) nor the Arduino (D0/D1) connectors). We will solve this inconvenience later.

### III.3.b. Validation with a serial terminal

🖥️ Open a serial terminal, any of your kind (Teraterm, PuTTY, GTKTerm, ...). Set it up so it can exchange data with the STLink VCP (baud rate, parity, stop bits, ...).

📄 You should see some text on the terminal. Check that this is the string you send with the STM32, and confirm by taking a screenshot of the serial terminal.



```

Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Dec 23 2019, 02:06:26.
Port /dev/ttyACM0, 18:25:39

Press CTRL-A Z for help on special keys

Hi
Hi

```

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyACM0

**Do not go further until this step has not been proved operational!**

The serial terminal is a very practical way of debugging and testing an application. In the embedded firmware development, it is the equivalent of a `printf("here");` in C.

### III.3.c. Analysis of the transmitted frame with a logic analyzer

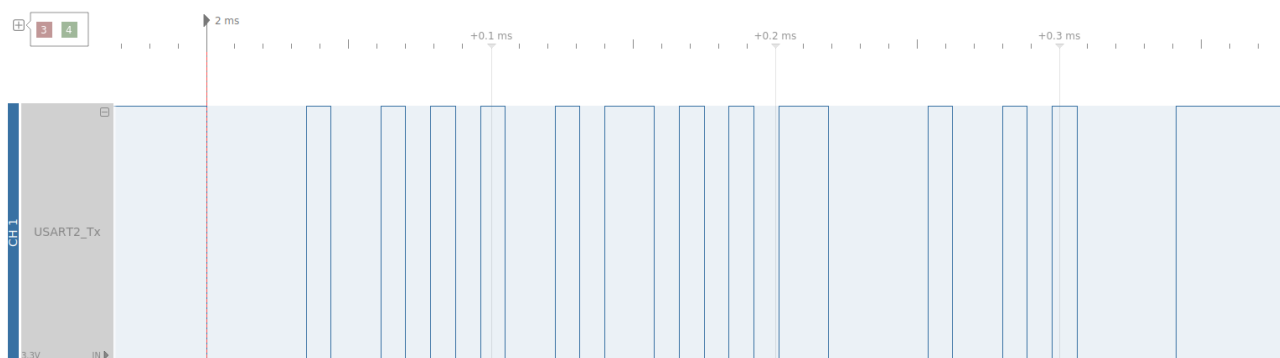
Now it is time to analyze the frame using a powerful tool: the logic analyzer. To do so you will have to probe the Rx and Tx electric signals. But we have seen that these pins are not linked to the NUCLEO connectors, but to the STLink. If we want to analyze those signals, we will have to find a way to their physical signals.

✍ In part III.1 Project creation page 24, you wrote the name of the Rx and Tx pins of the USART2 peripheral. Write them again here:

✍ Open the NUCLEO board schematic, given in the **datasheets** folder of the lab archive. On this schematic, find the target MCU, the Rx and Tx pins and follow the wires until you encounter a connector that is easily accessible with probes. To help you, list here every label you see, from the MCU pins to the STLink pins.

📁 Once you have identified the connector from which you will measure the Rx and Tx signals, make a capture with the logic analyzer. A tutorial for using the IKALogic analyzer and the ScanaStudio software exists in the **tutorials** directory.

✍ Here is a frame captured with the logic analyzer. It strictly is the signal that you are supposed to capture. Write the logic level of each bit, bit by bit, and find the original message.



📁 This work is quite tedious, even more when there is a large amount of captured framed waiting to be decoded. Instead of doing this again, use the frame decoding tool in ScanaStudio. Also display the HexView (go back to the tutorial if needed).

This time again, do not pass this point until you have decoded the frame using the logic analyzer tools.

### III.4. UART reception

#### III.4.a. Blocking reception

The simplest way of receiving data on the USART2 is to use the `HAL_UART_Receive()` function. Read the `stm32l0xx_hal_uart.c` file in which this function is defined.

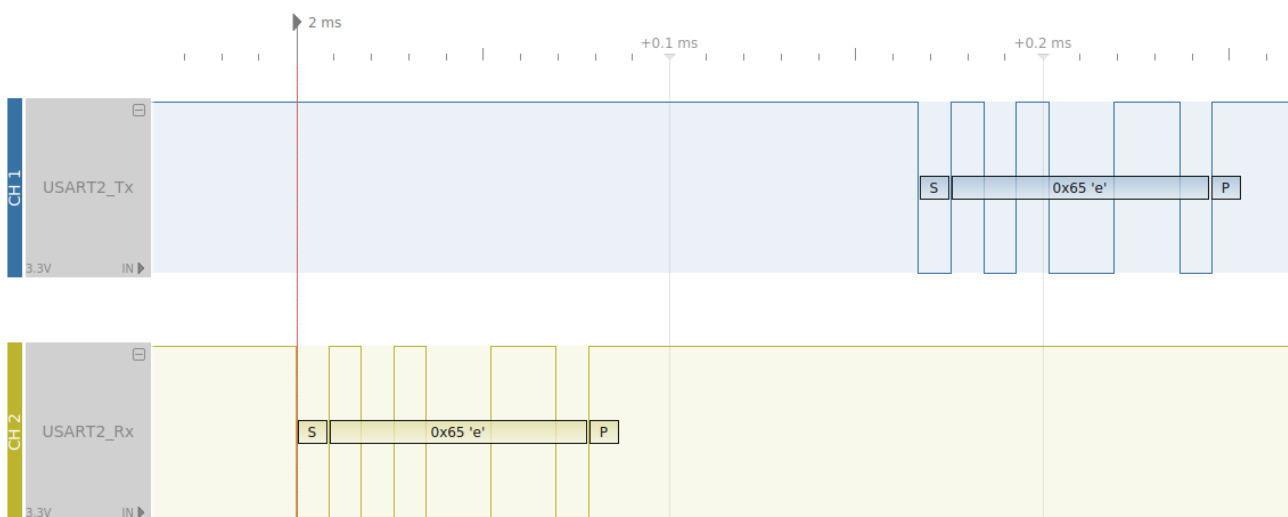
- ✎ What arguments should you pass to the `HAL_UART_Receive()` function?

📖 In the `main()` function, *after* the UART initialization and *before* the `while(1)` loop, send the string `"\r\nApplication starting\r\n"`. This will indicate on the serial terminal that the firmware has started running.

📖 In the `while(1)` loop now, wait for a character to be received and then send it back through the UART. This is called an echo application, which will replicate on the serial terminal the character that it just sent.

- 🔧 Confirm the operation with a serial terminal.

🔧 Confirm the operation with a logic analyzer capture. You should observe something similar to the figure below.



### III.4.b. Non-blocking reception

Although the `HAL_UART_Receive()` function is easy to use, its main drawback is that it uses polling. In other words it is a blocking function: it monopolizes the processor's CPU just to wait for a data to be received, preventing any other instruction to be executed.


To keep executing the firmware while waiting for data to arrive, we will use the interrupt mechanism thanks to the `HAL_UART_Receive_IT()` function.

Interrupts have been explained in Part 1 III.4 page 16 (read it again if necessary), so we will be short here. The interrupts of an ARM processor are controlled by the NVIC (*Nested Vector Interrupt Controller*). When a peripheral detects an event, it can throw an Interrupt Request (IRQ) if the NVIC has been configured this way. An Interrupt Service Routine (ISR) is then executed to process the event. However the ISRs are not seen by the developer (you). Indeed STM32CubeMX provides callback functions that are called by the ISRs. The developer only has to manipulate simple HAL functions.


Data reception on a STM32 UART follows this scheme. Don't panic, everything has been prepared by the STM32CubeMX initialization code generator when you asked for using the interrupt with the USART2 peripheral, during the project configuration.

The `HAL_UART_Receive_IT()` is a non-blocking function. It "only" configures an interrupt that will trigger when a character is received, then the function ends and the code execution goes on.

When one character is received on the specified USART, the CPU pauses the running execution and goes to the Interrupt Service Routine instead<sup>6</sup>. This ISR (which we will not modify) calls the `HAL_UART_RxCpltCallback()` callback function. This one is a function that **you have to define** by adding the processing you want to realize.

 Take a look at this callback declaration in the `stm32l0xx_hal_uart.c` file. What does the `__weak` qualifier mean?

<sup>6</sup> Function `UART_RxISR_8BIT()` defined in file `stm32l0xx_hal_uart.c`.

 You will code an echo application (any received character is sent back) by using the USART2 peripheral with interrupts. But first, you will create an application that counts every second and sends the time value to the UART. Both applications together will be merged into one, the default task (time stamp counter) being constantly running except when interrupted by the event task (echo).

1. Send a message to the computer only when the application starts
2. Send a message to the computer every second
  - This message is the value of a time stamp counter, incremented each second
  - Use the `sprintf()` function to create a string out of a number (or any variable)
  - Send this string to the computer, through UART
3. Before going further, confirm the operation of steps 1 and 2
4. Before the `while(1)` loop, call the `HAL_UART_Receive_IT()` function to enable the interrupt when ONE character has been received
  - You will have to define the character as a global variable to be able to use it in the callback
5. Between the `/* USER CODE BEGIN 0 */` and `/* USER CODE BEGIN 0 */` tags, write the definition of the `HAL_UART_RxCpltCallback()` callback.
  - It must instantly send back the received character
  - It must then activate a new reception with interrupt, so it can process the upcoming characters
6. Compile, upload, observe.

 Confirm with a screenshot of your serial terminal.



```

duboudier@dboudier-Precision-3541: ~
8

Application starting...
0
1
2
hello 3
4

Application starting...
0
1
again 2
3
bye 4
5
6
7
  
```

*Example of what you could observe on a serial terminal.*

### III.5. Bridge UART (FRENCH ONLY)

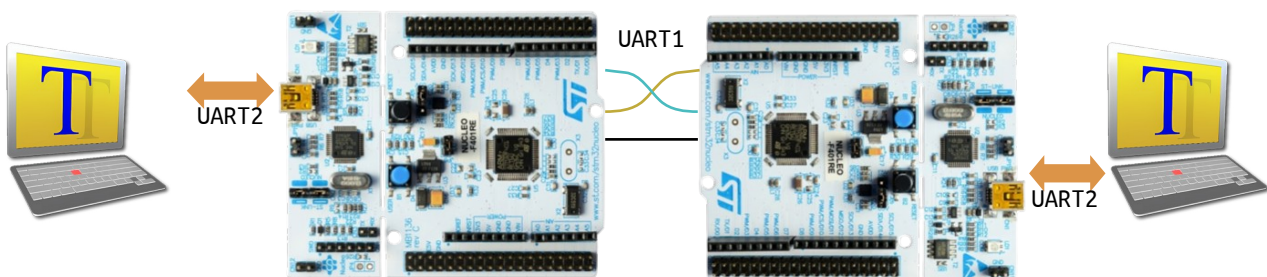
Vous allez dans cette partie créer un bridge UART. Dans le langage des bus de communication, un bridge est une interface qui ne fait que retransmettre un message venant d'une interface vers une autre, avec généralement une conversion de protocole entre les interfaces. Autrement dit, cela change la forme du message sans en changer le contenu.

Dans le cadre de cet exercice, aucune conversion de protocole ne sera effectuée. L'application bridge redirigera simplement le messages de l'UART2 vers l'UART1 d'un même MCU, et inversement.

#### III.5.a. Matériel

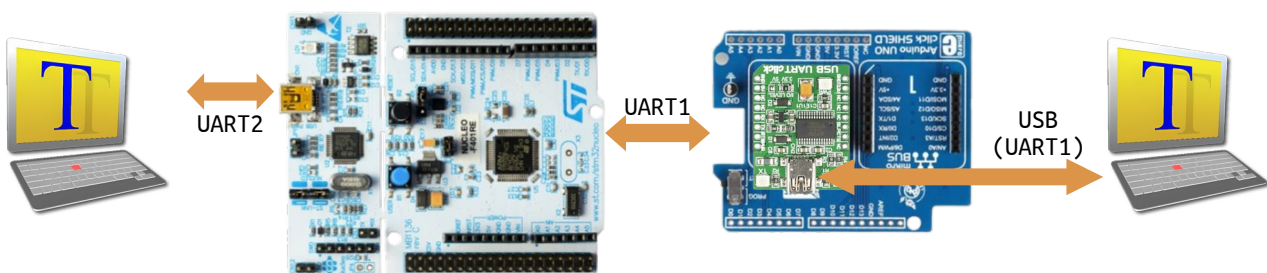
D'un point de vue matériel, vous avez le choix entre deux solutions.

La première solution nécessite plus de matériel, mais permet de mieux comprendre les connexions. En complément à votre NUCLEO, il faudra utiliser une deuxième NUCLEO (identique ou non) avec exactement les mêmes configurations de périphériques. Il faudra alors relier entre eux les périphériques UART1 de chaque NUCLEO (ne pas oublier la masse). Les deux NUCLEO sont reliées à un ou deux ordinateurs (mais bien deux terminaux série) via leur STLink VCP (UART2).



La deuxième solution est plus compacte puisqu'une seule NUCLEO suffit. Il faut néanmoins ajouter un autre convertisseur USB-UART afin de voir les messages de l'UART1. Pour cela vous pouvez réutiliser le USB UART click (vu en Systèmes Embarqués), en l'apposant sur un Arduino UNO click shield. Ainsi la NUCLEO sera reliée à un terminal série via son STLink VCP (UART2) et à un autre terminal série via le USB UART click (UART1).

Cependant les signaux Tx et Rx du périphérique USART1 ne sont pas directement routés sur les Click Board du shield. Il faut donc relier physiquement, à l'aide de fils, les broches de l'USART1 côté NUCLEO avec les broches de l'UART côté shield. Un peu de recherche avec le schéma électrique est nécessaire pour y parvenir ...





### III.5.b. Firmware

📖 Pour cette partie, créez un nouveau projet avec les propriétés suivantes :

- **Board selector** : celle que vous avez en TP (voir au dos, référence NUCLEO-xxxxxx)
- **Project Name** : VOTRENOM\_uart\_bridge ;
- **Project Location** : sélectionnez le répertoire `connectivity/workspace/uart_bridge/` ;
- **Initialize all peripherals with their default Mode ?** : Yes.

📖 Vous aurez besoin des informations suivantes lors de la **configuration** du MCU :

- Connectivity → USART2 **et** USART1
  - → Parameter Settings
    - Baud Rate = 115 200 baud
    - Bits de données = 8 bits
    - Bit de parité : Non
    - Bit de stop : 1 bit
  - → NVIC Settings
    - USART1/2 Interrupt : Enable (ou pas ! Cf cahier des charges ci-dessous)
  - → GPIO Settings
    - Relevez les broches utilisées et leur fonction dans le périphérique UART1
      - ✎ USART1\_TX = \_\_\_\_\_ USART2\_TX = \_\_\_\_\_
      - ✎ USART1\_RX = \_\_\_\_\_ USART2\_RX = \_\_\_\_\_

📖 Une fois la configuration effectuée : **Project → Generate Code** pour générer le code correspondant aux configurations des périphériques (ou l'icône 🛠️).

Le cahier des charges de votre application est en apparence assez simple :

- tout caractère reçu sur l'UART1 doit être renvoyé sur l'UART2 (bridge)
- tout caractère reçu sur l'UART2 doit être renvoyé sur l'UART1 (bridge)
- tout caractère reçu sur l'UART2 doit être renvoyé sur l'UART2 (echo)
- uniquement si vous utilisez une NUCLEO et un USB UART click :
  - tout caractère reçu sur l'UART1 doit être renvoyé sur l'UART1 (echo)

🖥️ En jouant sur avec les différentes fonctions de réception (réception bloquante par *polling* ou réception non-bloquante par interruption), répondez à ce cahier des charges.

🔧 Testez en lançant deux terminaux série et après avoir préparé le matériel.

✍️ Tracez un diagramme de séquence illustrant les données échangées, en partant d'un message issu d'un des deux terminaux.

## IV. Overview

✍ Go through all points of this part, and write here a summary of what you understood. This will be available for you if needed for future developments.

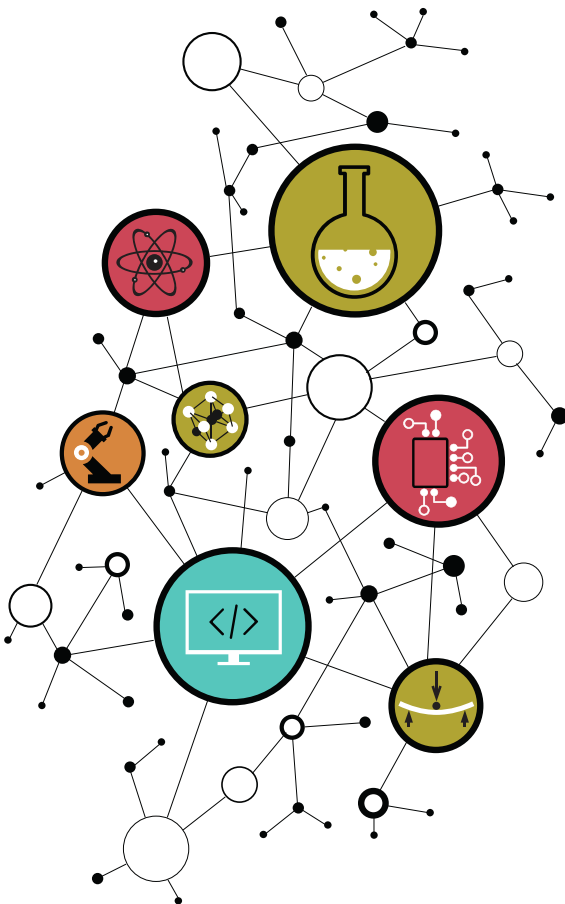
1. About the physical layer (wires, signals, voltage levels, logic levels, ...)
2. About the protocol layer (number of bits, bit coding, start/stop/parity bits, idle state, ...)
3. About the firmware layer (peripheral configuration, used functions)
4. About the application layer (serial terminal, logic analyzer).



PART 3

SPI –

# SERIAL PERIPHERAL INTERFACE



### I. Specifications

The **SPI (Serial Peripheral Interface)** is a serial communication bus initially designed by Motorola in the early 1980's. It is not a standard though and no organization has control of it. There are many variations around the original SPI version, but we will focus on the Motorola version as it is the most widespread version (some say it it a *de facto* standard).

The SPI is a **serial, synchronous** and **full-duplex** communication bus. It follows a **master-slaves** topology. This means the master is the only initiator of the data exchanges and the slaves are only authorized to answer to the master's requests.

The *master-slave* terminology is considered as not acceptable by some market players, due to its Historical weight<sup>7</sup>. Consequently those market players decided to change the SPI protocol terms by removing the *Master* and *Slave* words. However no organization has the power to define the SPI standard, so many market players made their own changes using their own terms. Nowadays you could see *Main, Controller, ...* to point out the *Master* and *Sub, Peripheral, Chip, Target, ...* to point out the *Slave*.

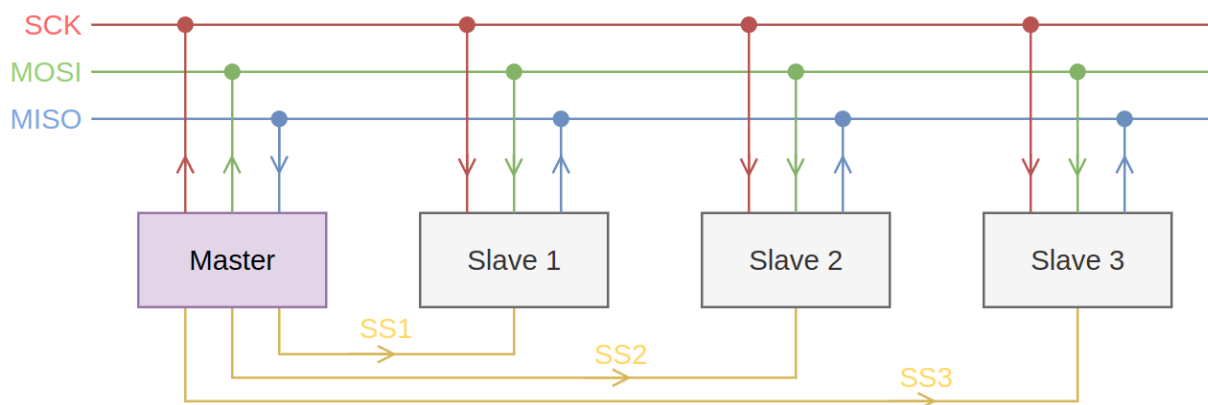
It has been decided for this course to use the original terminology, because (1) every electronic engineer knows these terms (whether they use it or not), because (2) not all online resources (websites, datasheets, tutorials, ...) have evolved (and if they did, they did not chose the same terms) and because (3) from a instructive view this makes a difference with the terms used in other serial communication protocols.

The SPI protocol uses three signals that are shared by all the bus users:

- **SCK** : Serial Clock sometimes SCL, SCK
- **MOSI** : Master Out, Slave In sometimes Main Out, Sub In
  - SDO (Serial Data Out), PICO (Peripheral In, Controller Out), COTI (Controller Out, Target In), ...
- **MISO** : Master In, Slave Out sometimes Main In, Sub Out
  - SDI (Serial Data In), POCI (Peripheral Out, Controller In), CITO (Controller In, Target Out), ...

Additionally the master holds a direct connection for every slave:

- **$\overline{SS}$**  : Slave Select sometimes nSS (not Slave-Select),  $\overline{CS}$  (Chip Select), CE (Chip Enable)



<sup>7</sup> [https://en.wikipedia.org/wiki/Master/slave\\_\(technology\)](https://en.wikipedia.org/wiki/Master/slave_(technology))

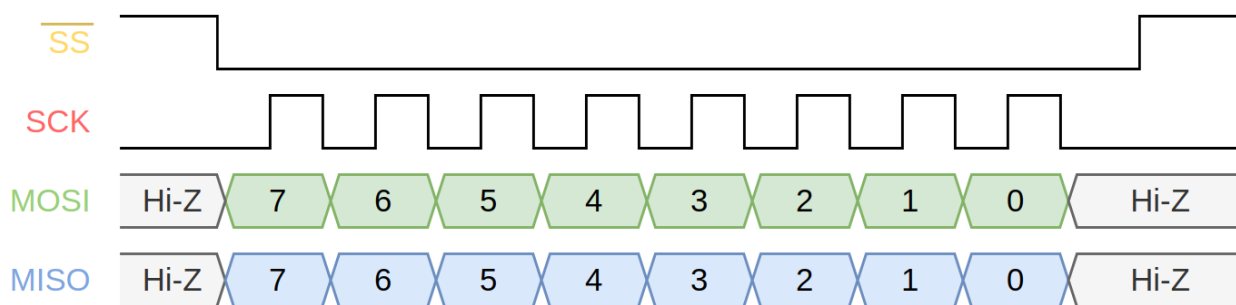
### II. Frame constitution

The SPI is not a uniformed standard, so any device manufacturer can adapt it its way. This page describes what a “classical” configuration looks like, even though you could meet counterexample when using others devices in the wild.

To start a data exchange with a slave, the master must first select it (or enable it) by setting a *usually* low logic state to the slave’s SS line.

Then the master provides a clock signal, usually of a 8-stroke duration. Each clock stroke let the master and the slave generate one bit, *usually* starting the the most significant bit (MSb).

In the *example* below, the master and the slave send their bit at each clock falling edge, and they read the input bit at each rising edge (letting time for MISO and MOSI lines to stabilize).



The clock idle state and the sample time (read time) of the bits are two parameters that are *usually* adjustable. They are respectively called **CPOL** and **CPHA**. A low-level idle state is noted CPOL = ‘0’, while a high-level idle state is noted CPOL = ‘1’. The value CPHA = ‘0’ means that the first bit sample instant will be on the first clock edge, while CPHA = ‘1’ means that the first bit sample instant will be on the second clock edge. The mode showed in the figure above is « Mode (0, 0) » or « Mode 0 »<sup>8</sup>.

Other variants exist but they will not be discussed here. Let us just cite the *three-wire SPI*. (because some of the sensors we will use support it). Being really close to the classical SPI (which is sometimes called *four-wire SPI*), the three-wire SPI only has one line dedicated to data exchange (SISO or MIMO), but is a bidirectional line. The *three-wire SPI* is a *half-duplex* version of the SPI, encountered in even lower power applications.

You got it, many parameters change according to the device manufacture, either on the master side or on the slave side.

*Usually* external peripherals (sensors, actuators, ...) are in a configuration mode that is set at the device fabrication. The MCU must adapt (by adjusting the configuration of its SPI peripheral) to ensure a working communication.

<sup>8</sup> No need for more explanations for this course. However the Wikipedia article is good basis if needed: [https://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface#Clock\\_polarity\\_and\\_phase](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface#Clock_polarity_and_phase)

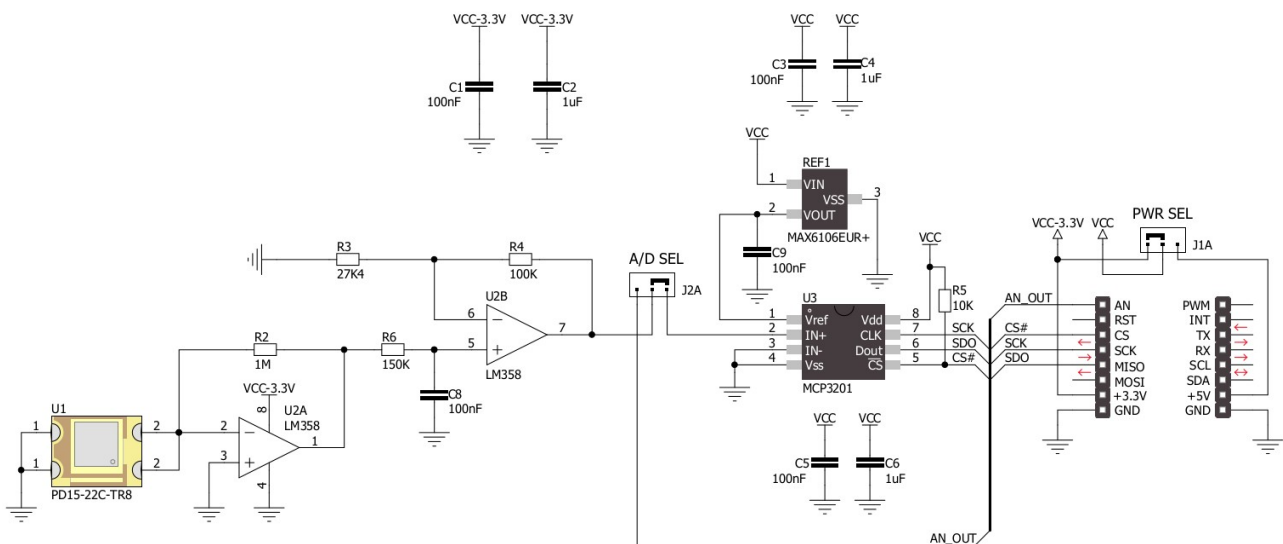
### III. Application: Light click

The first device you will communicate with is a Click board™ from the MIKROE company: the Light click<sup>9</sup>. It is a daughterboard that that is equipped with a luminosity sensor.

#### III.1. Daughterboard study

For every device we will use, you can find the useful documents (schematics, datasheets, ...) in the [connectivity/datasheets/](#) folder of the lab archive.

- ✎ From those documents, circle out the functional blocks of this board and their function.



- ✎ Find and write down the Click board pins that will be used for the SPI communication with the MCU.


- ✎ From the Arduino UNO click shield schematic and the NUCLEO board schematic, find which MCU pins are connected to the Light click SPI pins (beware, this depends on the slot in which you will place the Click board).

9 <https://www.mikroe.com/light-click>



### III.2. SPI peripheral configuration

You have seen that the MCP3201 is an ADC with a SPI interface. To exchange data with this ADC (and the Light Click), the MCU SPI peripheral must be configured too.



 In STM32Cube IDE, create a new STM32 project (go back to the tutorial if needed).

 You will need the following information while **creating** the project:

- **Board selector**: use the NUCLEO board reference (bottom side of the board);
- **Project Name**: YOURNAME\_spi;
- **Project Location**: select the directory `connectivity/workspace/spi/`;
- **Initialize all peripherals with their default Mode ?**: Yes.

 You will need the following information while the MCU **configuration** step:

- Connectivity → SPI1
  - Mode : Full-Duplex Master
  - Hardware NSS<sup>10</sup> Signal : Disable
  - Basic parameters : Motorola // 8 Bits // MSB First
  - Clock parameters : Prescaler = 2 // Clock polarity = Low // Clock Phase = 1 Edge
  - GPIO settings : use your answers to the previous questions to map the SPI1\_SCK // SPI1\_MISO // SPI1\_MOSI signals to the pins you have noted.
- System Core → GPIO
  - use you answers to the previous questions to map the CS/SS signal to its pin.
    - On the device view: left-click on the pin → **GPIO\_Output**
    - On the « GPIO Mode and Configuration » tab, P<sub>??</sub> configuration :
      - GPIO Output level = High
      - ...
      - User Label = LIGHT\_nSS


 Generated the code (  ) and check that the `main()` function contains a call to the `MX_SPI1_Init()` function/

The STM32CubeMX initialization code generator shows one again its use, as the SPI peripheral is ready for use in few minutes. To compare with the USART peripheral, the “SPI/I2S” chapter of the *STM32L0x3 user manual* counts 50 pages (for a total of 1,040). The peripheral itself uses nine registers.

<sup>10</sup> NSS = Not Slave Select =  $\overline{SS}$ , or also NCS = Not Chip Select =  $\overline{CS}$

### III.3. light\_click driver

Of course the SPI peripheral is configured and waiting to be used, but we will first write driver functions that will specifically address the Light click through the SPI bus. A draft version of the driver files has been written and is provided in the lab archive. You will first import them and then complete them.


 The driver files are located in the `connectivity/workspace/drivers/` folder. Copy both files `light_click.c` and `light_click.h` and paste them into the project workspace, i.e. the `.../CM4/Core/Src/` and `.../CM4/Core/Inc/` folders, respectively. From the IDE project explorer, you may need to refresh (F5) these two folders for the new files to appear.


At first the header file seems “rather” simple to use: there is a initialization function and two read functions. This is because the MCP3201 is also rather simple: it is not configurable and the only thing it does is sending a converted value to the SPI MISO line. In the header file you will also find a `LIGHT_handle_t` structure. To keep it simple a **handle** is an object that contains information (configuration parameters, state, ...) of a resource.

Let us get to the C file now. It contains the definitions of the three functions discussed above. But these definitions are empty, you will complete them step by step.

#### III.3.a. Handle initialization

Read the `light_click.h` header documentation (Doxygen comments).

 What is the purpose of the `hspi` field in the `LIGHT_handle_t` structure ? Which pins (or which signals) can it control?

 What is the purpose of the `SS_Port` and `SS_Pin` fields in `LIGHT_handle_t`? Which pin (or which signal) can it control?

 What is the purpose of the `LIGHT_init()` function ? Which parameters does it need?

✎ Browse the `main.c` and `main.h` files to find the SPI peripheral handle and the definition of the SS pin number and port. What are their names?

✎ In the `light_click.c` file, find the name of the Light click handle.

📄 Complete the `LIGHT_init()` function definition, according to the comments.

📄 In the `main()` function, write the code below so that the Light Click initialization function is called. Remember that the function documentation in `light_click.h` gives you hints about the arguments needed by this function.

```

/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart2, (uint8_t*)"App initialization...\r\n",
                  strlen("\r\nApp initialization...\r\n"), HAL_MAX_DELAY);
LIGHT_init( /** @TODO */ );
HAL_UART_Transmit(&huart2, (uint8_t*)"App running...\r\n",
                  strlen("App running...\r\n"), HAL_MAX_DELAY);
/* USER CODE END 2 */

```

📄 Compile. If there is any error or warning, fix your code. Repeat until everything is fine.

Note that it is no use executing the program now as you have not written the read function yet.

### III.3.b. Read function

Let us switch now to the `LIGHT_readBrightnessRaw()` function development. This function is the simplest of the two read functions.

✎ Read section « 5.0 Serial communications » of the MCP3201 datasheet. Which signals are necessary and what are the conditions to fulfill in order to trigger a conversion read out?

✎ Write the number of clock strokes necessary for reading a complete data. What happens when the clock still beats after a complete data has been received?

✎ How should we do to read a complete acquisition, knowing that a classical SPI peripheral can only sends 8-stroke long clock signals<sup>11</sup>? The answer is in section « 6.1 Using the MCP3201 Device with Microcontroller SPI Ports ».

✎ What operations should we perform on the received data to make it a 12-bit value?

---

**11** It is actually possible to configure this SPI peripheral to use frame ranging from 4-bit to 16-bit. We keep a 8-bit size to stay close to the original SPI version (and to remain compatible with the next Click boards).

✎ What is the name blocking read function of the SPI peripheral? What arguments does it need?

📖 With your previous answers, complete the `LIGHT_readBrightnessRaw()` function so it triggers a two-byte read out, and then converts them into a 12-bit integer value. The latter will be returned through the `*brightness_raw` pointer.

📖 In the `while(1)` loop of the `main()` function, add the following code.

```
HAL_Delay(1000);

// LIGHT CLICK VALIDATION
LIGHT_readBrightnessRaw( &brightness_raw );
sprintf( uart_out, "LIGHT\t Raw brightness = 0x%04X = %5d\n\r", brightness_raw,
brightness_raw );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

📖 Compile, fix the errors and upload the firmware into the target MCU.

🔧 Open a serial terminal and observe the result. Does the displayed value seem correct?

It is high likely that the MCU only received `0x0000`'s. If that is the case, this means the slave (the Light click) does not respond to the master requests. We need to use a logic analyzer to understand that better.

### III.3.c. Decoding the exchanged frames

🔧 Use the logic analyzer to probe the four SPI signals. You have previously noted the pin number of these signals and you also have the NUCLEO board schematic.

✎ With a simple capture you should observe two eight-stroke clock signals. This means the master MCU is actually asking for a read out but the slave does not seem to react. Which signal should be sent to the slave so that it knows it should process the request? Did you forget it?

✎ This signal is controlled by a GPIO pin configured as a `GPIO_Output`. Which function should you use to drive a GPIO? (hint: you used it to control the LED).

☞ Complete the `LIGHT_readBrightnessRaw()` function definition so that the slave is selected before asking for a read out, and then the slave is deselected.

☞ Compile, upload into the MCU and confirm with the logic analyzer that the slaves does answer to the master requests. Make a screenshot of this capture.

☞ Also validate with a serial terminal that the luminosity value is valid and varies.

### III.3.d. Floating point read function

We could have decided to take every Light click component into account and get a value more physically accurate than just an integer value between 0 and 4095. Yet we want to keep it simple and we will only ask for an irradiance value in percentage. This is the objective of the `LIGHT_readBrightnessPercentage()` function.

☞ Complete the definition of `LIGHT_readBrightnessPercentage()` so it matches the requirements written in the Light click header file.

☞ In the `main()` function infinite loop, add the following lines.

```
LIGHT_readBrightnessPercentage( &brightness_percent );
sprintf( uart_out, "LIGHT\t Brightness percentage = %.2f%\n\r", brightness_percent );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

Note: it is likely that your toolchain shows you this warning message.

« *The float formatting support is not enabled, check your MCU Settings from "Project Properties > C/C++ Build > Settings > Tool Settings", or add manually "-u \_printf\_float" in linker flags.* »

Follow this advice to remove the warning but mostly to be able to send floating-point numbers to the serial terminal.

☞ Confirm the operation with a serial terminal.

Congratulations! You have just reached a milestone in your embedded development skills by developing a driver for a device communicating through SPI!

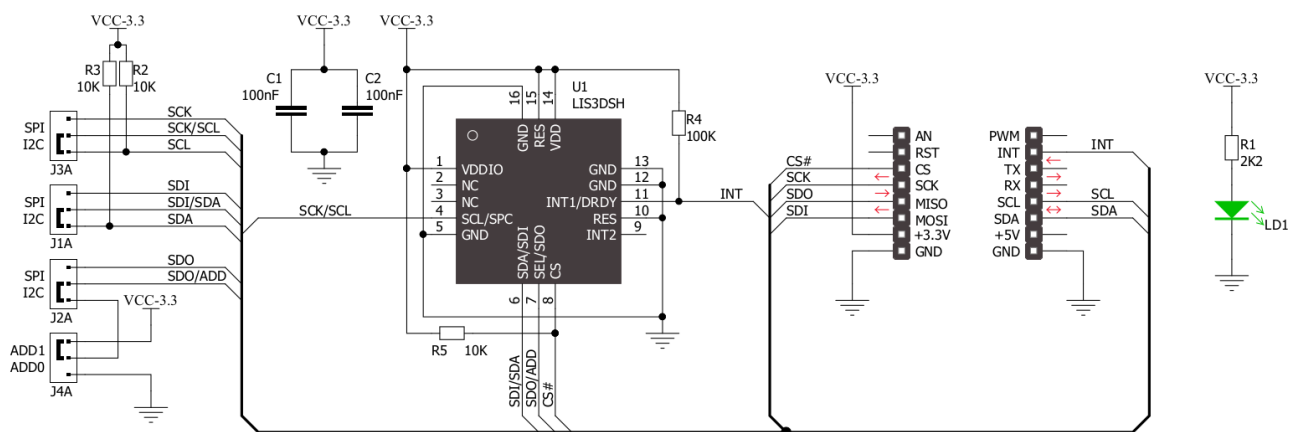
### IV. Application: Accel 2 click

We will now use another Click board™ : the Accel 2 click<sup>12</sup>. As you can imagine this is an accelerometer. This device communicates using an SPI interface, just like the Light click, but you will see that this one is more complex.

#### IV.1. Sensor study

In this case studying the schematics shows no interest as the Accel 2 click contains only one Integrated Circuit (IC): the 3-axis LIS3DSH accelerometer from STMicroelectronics. Still we can see the presence of four jumper connectors on the left of the schematics.

✎ To which physical parts on the board does these jumper correspond to? What is their function?



✎ Plug the Accel 2 click into an *Arduino UNO click shield* slot and note to which MCU pins are connected the MOSI, MISO, SCK and CS signals.

<sup>12</sup> <https://www.mikroe.com/accel-2-click>


### IV.2. SPI peripheral configuration

You should have noted that there is no need to reconfigure the SPI peripheral as the Accel 2 click uses the same pins as the Light Click. Indeed that is the point of using a communication bus.

**The MOSI, MISO and SCK signals are common to the master and all the slaves.**

However the CS signal is unique for each slave. So we still need to set the GPIO that drives the Accel 2 click's CS signal.

- 📄 Carry on the `YOURNAME_spi` STM32CubeIDE project.
- 📄 Open the `YOURNAME_spi.ioc` file so you can configure the CS pin.
- System Core → GPIO
  - On the pinout view: left click on the pin → `GPIO_Output`
  - On the “GPIO Mode and Configuration” tab, P<sub>??</sub> configuration:
    - GPIO Output level = High
    - ...
    - User Label = `ACCEL_2_nSS`


📄 Generate the code (  ).

You should observe that the `MX_GPIO_Init()` function (in `main.c`) now contains new code lines, in charge of this GPIO initialization.



### IV.3. accel\_2\_click driver

Let us deal the the Accel 2 click driver.


 The driver files are located in the `connectivity/workspace/drivers/` folder. Copy the `accel_2_click.c` and `accel_2_click.h` files and paste them into `.../CM4/Core/Src/` and `.../CM4/Core/Inc/` project folders, respectively. From the Project explorer tab (left side of the IDE) you might need to refresh (F5) both folder to see the new files appear.


At first the header file seems much longer than the Light click header. But with a deeper looks the first part only contains macro-constants (which will be studied later). And the following part is rather close to the previous driver: we find a structure declaration (`ACCEL_2_handle_t`, which contains the Accel 2 click initialization parameters), an initialization function, few read functions and a write function (all listed below).


```
void ACCEL_2_init(SPI_HandleTypeDef *hspi, GPIO_TypeDef *SS_Port, uint16_t SS_Pin);
void ACCEL_2_writeReg(uint8_t reg_addr, uint8_t reg_value);
void ACCEL_2_readReg(uint8_t reg_addr, uint8_t* reg_value, uint8_t n_bytes);
void ACCEL_2_readX(float* x_value);
void ACCEL_2_readY(float* y_value);
void ACCEL_2_readZ(float* z_value);
void ACCEL_2_readXYZ(float* x_value, float* y_value, float* z_value);
```

In the C file, the functions definitions are still yours to complete. We will write them step by step.

#### IV.3.a. Handle initialization

 Browser the `main.c` and `main.h` files to find the SPI peripheral handle and the SS port number and pin number. What are their names?

 Start to write the definition of the `ACCEL_2_init()` function (comment `@TODO (1)`) by initializing the Accel 2 handle with the function parameters.

 In the `main()` find the adequate location to call the Accel 2 click initialization function. Compile and fix your code until there is no warning and no error.

It is no use to upload the firmware onto the target MCU as the read and write function have not been completed yet.

### IV.3.b. Accel 2 click read function

✎ From the chapter “5 Digital interfaces” of the LIS3DSH datasheet, three communication modes are available. Which are they?

✎ Datasheet’s table 8 show that the signals name is different from the classical one (CS, SCK, MOSI, MISO). Indeed the signals name can change with the manufacturer’s habit. Note here the translation if necessary.

✎ Like all self-respecting datasheets, this one explains how to access to a data stored in the device. Find these information and write them down here.

📄 From the previous answer, complete the definition of the `ACCEL_2_readReg()` function. You can also read the Doxygen comment block that documents this function (in the `.h`).

📄 In the `while(1)` loop of the `main()` function, add the lines below. Compile and fix the errors and warnings.

```
ACCEL_2_readReg( ACCEL_2_REG_WHO_AM_I, &reply, 1 );
sprintf( uart_out, "ACCEL\t Who am I = 0x%02X\n\r", reply );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

✎ What does this code does? What value should be displaying onto the serial terminal?

📄 Confirm with a serial terminal screenshot.

☞ Also make a screenshot of a frame captured by a logic analyzer. Compare the frame with “Figure 8. SPI read protocol” of the datasheet. You should observe some differences. Let us explain them with the next questions.

✎ According to the datasheet, what should be the idle state of the SCLK line? What is the idle state on your captured frame?

✎ According to the datasheet, when do the MOSI and MISO lines should sample their input bits? What do you observe on your captured frame.

✎ According to the datasheet, when do the MOSI and MISO lines should send their output bit? What do you observe on your captured frame?

These information correspond to the **(CPOL, CPHA) modes**. The CPOL (*Clock POLarity*) parameter indicates the clock line idle state ('0' = low level, '1' = high level). The CPHA (*Clock PHase*) parameter indicates if the MOSI/MISO bits are sampled on the first edge ('0' value) or on the second edge ('1') of the clock signal.

Thus modes (0, 0) and (1, 1) are compatible for exchanging data because in both cases the bits are output on the clock falling edge and they are read on the clock rising edge. The clock idle state is the only difference. Similarly (0, 1) and (1, 0) modes are compatible together.

✎ To which mode corresponds the section “5.2 SPI bus interface” of the LIS3DSH datasheet?

✎ In which mode has our SPI1 peripheral been configured? The answer is in the `.ioc` file.

Conclusion: configured this way the MCU SPI peripheral is compatible with the LIS3DSH, even though this is not exactly mode required by the accelerometer.

By the way, if you go back to the Light Click ADC, the MCP3201 datasheet says: “*SPI Mode 0,0 (clock idles low) and SPI Mode 1,1 (clock idles high) are both compatible with the MCP3201*”, proven by the figures 6-1 and 6-2 that presents both modes.

### IV.3.c. Accel 2 click write function

The LIS3DSH possesses many registers and some of them contain the sensor's configuration parameters. We will have to set them to specific values in order for the LIS3DSH to work according to our needs.

✎ Find in the datasheet the information explaining how to modify a value stored in a register. Write them down here.

- 📄 Complete the definition of the `ACCEL_2_writeReg()` function.
- 📄 Confirm by adding the following code to the `while(1)` loop, in the `main()`.

```
ACCEL_2_writeReg(ACCEL_2_REG_VFC_1, 0xAB);
ACCEL_2_readReg(ACCEL_2_REG_VFC_1, &reply, 1);
sprintf( uart_out, "ACCEL\t VCF_1 = 0x%02X\n\r", reply );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );

ACCEL_2_writeReg(ACCEL_2_REG_VFC_1, 0x00);
ACCEL_2_readReg(ACCEL_2_REG_VFC_1, &reply, 1);
sprintf( uart_out, "ACCEL\t VCF_1 = 0x%02X\n\r", reply );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

✎ What are the operations performed by those lines? (*Note: you are not asked to explain the role of the VFC\_1 register: it will not be used later and its value has no effect of the sensor's operation in our case*).

- 📄 Confirm the operation of the `ACCEL_2_writeReg()` function with a serial terminal.
- 📄 Capture with a logic analyzer the write operation followed by the read operation.

### IV.3.d. LIS3DSH configuration

Now that the SPI communication functions between the MCU and the Accel 2 click are operational, we will be able to configure the LIS3DSH so it matches our requirements. This is the main difference with the previous sensor (Light click) that was no configurable.

Look at chapter “6 Register mapping” of the datasheet. There is a table that lists all the registers, with their address, their access mode (r, w, r/w), their description and maybe their default value (i.e. at power on). If you compare this datasheet’s table with the first part of the `accel_2_click.h` file, you will find out that the list of macro-constants defines all the registers by their address so that they can be used with the C driver.

When an experimented developer opens the datasheet of a new sensor, he will always look for the registers named `CTRL`, `CFG`, `SETTINGS`, or anything similar. Those register often contains the adjustable parameters of the device. With few back-and-forths in within the datasheet, he can obtain a configuration that does the trick. Then a detailed datasheet reading process is necessary to set the parameters in order to match perfectly with the requirements (and maybe optimize its power consumption, ...). However being able to fully configure a sensor is not the prime objective of this course. As a consequence the registers and parameters identification phase is prepared by the subsequent questions.

✎ Put yourself in a expert developer’s shoes and find the registers that should help you configure the LIS3DSH (there are 6 or 7 of them).

After reading the description of those registers, a quick look at it should let us extract the *interesting* bits (the highlighted words are the keywords the expert developer looks for).

ODR3	ODR2	ODR1	ODR0	BDU	Zen	Yen	Xen
------	------	------	------	-----	-----	-----	-----

#### register description

ODR 3:0	Output data rate and <b>power mode</b> selection. Default value: 0000 (see <a href="#">Table 55</a> )
BDU	Block data update. Default value: 0 (0: continuous update; 1: output registers not updated until MSB and LSB have been read)
Zen	Z-axis <b>enable</b> . Default value: 1 (0: Z-axis disabled; 1: Z-axis enabled)
Yen	Y-axis <b>enable</b> . Default value: 1 (0: Y-axis disabled; 1: Y-axis enabled)
Xen	X-axis <b>enable</b> . Default value: 1 (0: X-axis disabled; 1: X-axis enabled)

**ODR[3:0]** is used to set the **power mode** and ODR selection. In [Table 55](#) (output data rate selection) all available frequencies are shown.

- ✎ Find the previous register and explain the value that should be written to its bits.

Before actually configuring a device, it is use to check if it is reachable through the SPI bus. This is the role of the `WHO_AM_I` register that contains a predefined value: by reading this register and comparing it to its expected value, one can confirm (or disconfirm) the sensor's operation.

- 📖 Complete the definition of `ACCEL_2_init()` following the `@TODO (2)` comment (checking the sensor's presence) and the `@TODO (3)` comment (sensors configuration).

- 📖 Compile, upload into the target.

- 🔧 Confirm with a serial terminal that the firmware does not continue if the sensor is not plugged in when starting the application.

### IV.3.e. Read the acceleration values

We will focus on reading the z-axis acceleration value by defining the `ACCEL_2_readZ()` function. If it is successful we will only have to replicate the code for the other read functions.

- 📖 First add the following code to the `while(1)` loop, in the `main()` function. This will let us test the z-axis read function on a regular basis (even if it is not operational yet).

```
float accel_z = 0.0;
ACCEL_2_readZ( &accel_z );
sprintf( uart_out, "ACCEL\t z-axis = %f g\n\r", accel_z );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

- 🔧 Compile, upload and observe the result on a serial terminal. The `ACCEL_2_readZ()` function's body is empty, so it is expected to always print `0.000000` on the terminal. At least we will keep track of the read function developments.

As it happens, let us switch to the `ACCEL_2_readZ()` function's development.

- 📖 `@TODO (1)`: from the datasheet, determine which register(s) should be accessed to retrieve the z-axis acceleration value.

- 📖 `@TODO (2)`: from the retrieved data, build a floating-point number. It can be useful to use an intermediate 16-bit integer variable.

- 🔧 Compile, upload and observe the result on a serial terminal. This time the displayed acceleration value should vary, especially when moving the board.

Of course the displayed value fluctuates with the board orientation, but it does not fit with acceleration units (neither in  $m \cdot s^{-2}$  nor in  $g$ ). The sensor's datasheet gives us the following information:

**Table 3. Mechanical characteristics**

Symbol	Parameter	Test conditions	Min.	Typ. <sup>(1)</sup>	Max.	Unit
FS	Measurement range <sup>(2)</sup>	FS bit set to 000		±2.0		g
		FS bit set to 001		±4.0		
		FS bit set to 010		±6.0		
		FS bit set to 011		±8.0		
		FS bit set to 100		±16.0		
So	Sensitivity	FS bit set to 000		0.06		mg/digit
		FS bit set to 001		0.12		
		FS bit set to 010		0.18		
		FS bit set to 011		0.24		
		FS bit set to 100		0.73		

✎ What is the meaning of both parameters?

✎ What are the **FS bits** in the test condition? What are their values in our case?

✎ How can that help us to obtain an acceleration value in  $g$  unit?

📄 **@TODO (3)**: build the **z\_value** number in the required unit ( $g$ ).

🔧 Compile, upload and observe the result on a serial terminal. Confirm that the acceleration value ranges from  $-1.0 g$  and  $+1.0 g$ , depending on the board orientation.

### IV.3.f. Adjusting: full-scale

With all the previous work, you have developed a driver that matches basic requirements, but at least it is operational. The LIS3DSH offer various functionalities (such as state machines that could be used for VR headsets, motion-controlled interface, ...). This is not this course's goal to make a demonstration of this sensor's full capabilities. However we can still refine some details to show you how it should be done.

📄 Let us clean the `while(1)` content (`main()`):

- Change the loop period to 100 ms (instead of 1 s);
- Comment or remove all the other lines, except for the z-axis readout and display.

🔧 Confirm that the acceleration value is displayed in the serial monitor.

🔧 Shake the target: you should see that the acceleration value saturates.

🔪 Which register and which bits can change the sensor's sensitivity? What should be their values so that the sensor works on the largest range?

📄 Complete the definition of the `ACCEL_2_init()` function by setting the new full-scale value (`@TODO (4)` comment).

🔧 Compile, upload, shake, confirm.

Oopsie, the acceleration value is not good anymore! As a matter of fact, the acceleration read function depends on the sensitivity value, which relies on the full-scale. Instead of writing a function that is valid for one sensitivity value only (thus for one full-scale value only) we will modify its definition to make it adaptable.

📄 In the definition of `ACCEL_2_readZ()`, remove the code that corresponds to the `@TODO (3)`.

📄 At `@TODO (4)`, perform a full-scale readout.


📄 At `@TODO (5)`, perform a test on the full-scale value and apply the corresponding sensitivity coefficient to the retrieved acceleration value.

🔧 Compile, upload, try and confirm.

📄 If the test application is valid, complete and confirm the other read functions, for the other axes: `ACCEL_2_readX()`, `ACCEL_2_readY()` and `ACCEL_2_readXYZ()`.



### V. Overview

 Go through all points of this part, and write here a summary of what you understood. This will be available for you if needed for future developments.

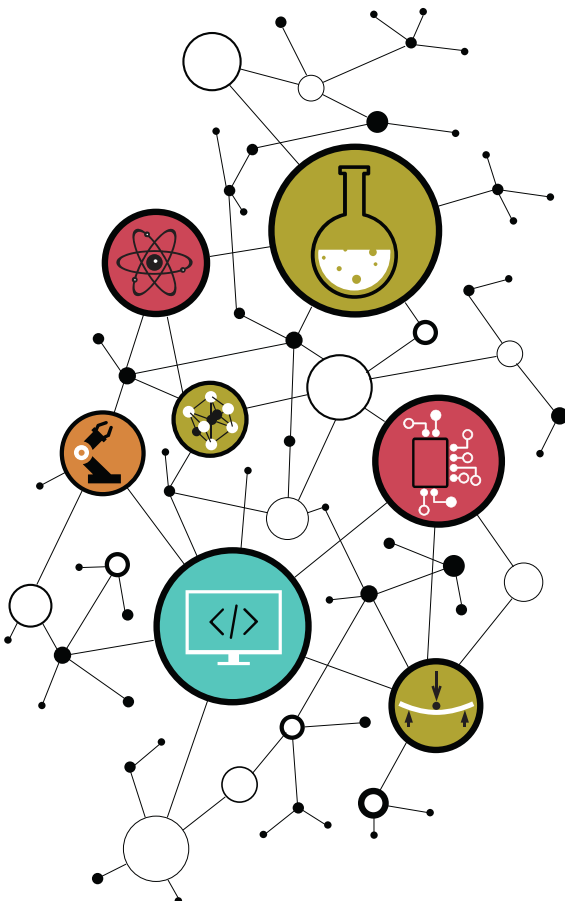
1. About the physical layer (wires, signals, voltage levels, logic levels, ...)
2. About the protocol layer (number of bits, who owns each line, ...)
3. About the firmware layer (peripheral configuration, used functions)
4. About the sensor layer (parameters, registers access, ...).



PART 4

I<sup>2</sup>C –

# INTER-INTEGRATED CIRCUIT



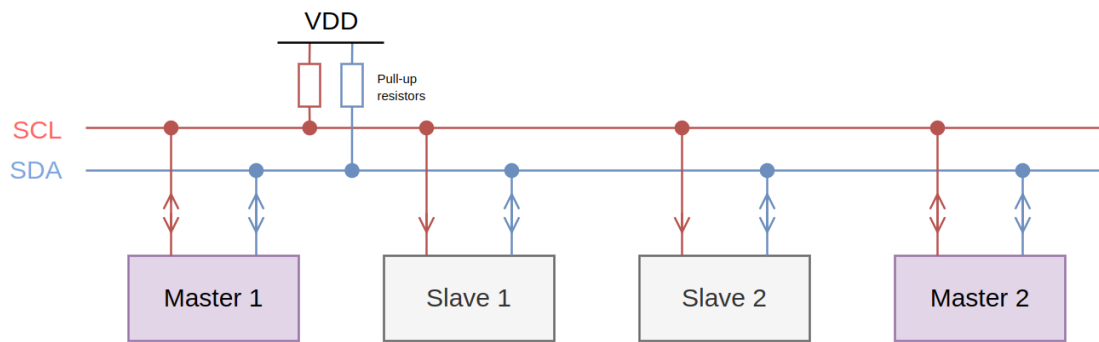
### I. Specification

Developed in 1992 by Philips Semiconductors (now NXP), the I<sup>2</sup>C bus is a **serial, synchronous, half-duplex** communication bus that uses a **master-slaves** topology. Contrary to SPI, I<sup>2</sup>C offers the possibility of having multiple masters on the bus.

The I<sup>2</sup>C bus is made out of two bidirectional lines:

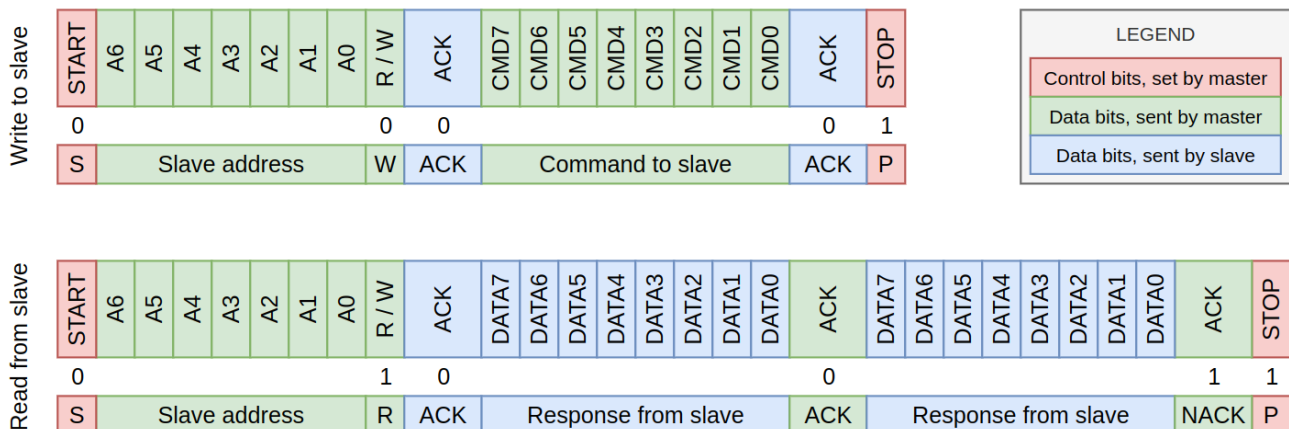
- **SCL**: Serial Clock, only controlled by the master nodes;
- **SDA**: Serial Data, controlled sometimes by a master node, sometimes by a slave node.

Both lines are pulled-up to a high logic state with two pull-up resistors.



The main advantage of I<sup>2</sup>C (when compared to SPI) is its very small need of wire: **only two lines are necessary**, while SPI requires  $3+n$  wires ( $n$  being the number of slaves). As a matter of fact, the slave selection with I<sup>2</sup>C is not made with an electric line but with an address frame. This leads to the drawback of I<sup>2</sup>C: it is **slower than SPI** (effective data rate).

To exchange data with a slave, the master first sends a start bit ('0'), 7 address bits for selecting the slave, an access mode bit (Read/Write), then lets the slave acknowledge ('0'). Then depending on the access mode (Read/Write), the SDA line will be controlled either by the master or the addressed slave. In both cases, the master is always the one that decides to stop the exchange with a stop bit (and, in read mode, the master does not acknowledge the last frame).

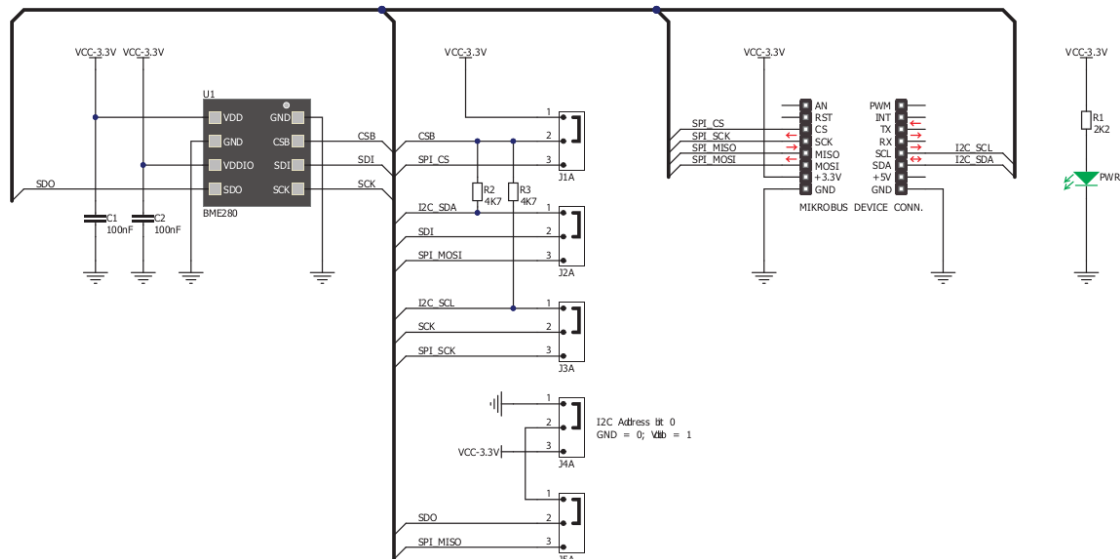


### II. Application: Weather click

Let us study the I<sup>2</sup>C protocol through another Click Board: the Weather click.

#### II.1. Sensor study

Once again, the Click board is just a simple connector interface for the embedded sensor (a Bosch Sensortec BME280), as you can see with this very simple schematics.



✎ Which physical quantities can the BME280 sensor measure (see its datasheet)?

✎ On this board, which signals are necessary for an I<sup>2</sup>C communication?

✎ Using the *Arduino UNO click shield* schematics and the NUCLEO board schematics, find the MCU pins to which are connected the I<sup>2</sup>C signals of the Weather click.

✎ Find the J4A jumper on the schematics and note its position (0 or 1) on the board.

✎ What is the purpose of the R2 and R3 resistors? Hints can be found in section “6.2 I<sup>2</sup>C interface” of the BME280’s datasheet.

### II.2. I<sup>2</sup>C peripheral configuration

As you could have seen, the communication interface of the Weather click is an I<sup>2</sup>C bus. Let us configure the I<sup>2</sup>C peripheral on the STM32 microcontroller.

📖 With STM32CubeIDE, create a new STM32 project (read the tutorial if necessary).

📖 You will need the following information while **creating** the project:

- **Board selector**: use the NUCLEO board reference (bottom side of the board);
- **Project Name**: YOURNAME\_i2c;
- **Project Location**: select the directory `connectivity/workspace/i2c/`;
- **Initialize all peripherals with their default Mode ?**: Yes.

📖 You will need the following information while the MCU **configuration** step:

- On the device view: select the I<sup>2</sup>C mode on the two pins marked previously.
  - That will make you identify the I<sup>2</sup>C peripheral to be used (I2C1, I2C2 or I2C3)
- Connectivity → I2Cx
  - I2C : I2C
  - anything else by default

📖 Generate the code (🔧) and check that the `main()` function now contains a call to the `MX_I2Cx_Init()` function.

### II.3. weather\_click driver

Just like the Click boards that have been previously used in this course, we will develop a driver so that any user will be able to exchange information with the Weather click. The driver files are located in the `connectivity/workspace/drivers/` directory.

📖 Copy the `weather_click.c` and `weather_click.h` files and paste them into the `.../CM4/Core/Src/` and `.../CM4/Core/Inc/` project folders, respectively. In the Project explorer (left side of the IDE), you may need to refresh (F5) the folders to see the new files appear.

Reading these files you should be now familiar with their contents. Only this time you can see in the `weather_click.c` file that some functions are declared and defined with the `static` keyword.

- 🔪 What does the `static` qualifier mean for a function and in C language?

### II.3.a. Handle initialization

Read the documentation (Doxygen comment blocks) of the `weather_click.h` file.

✎ What is the purpose of the `hi2c` field of the `WEATHER_handle_t` structure? Which pins (or which signals) can it control?

✎ Which parameters should we pass to the `WEATHER_init()` function?

✎ In the `main.c` file, find the declaration of the I<sup>2</sup>C peripheral's handle? What is its name?

✎ You only have found its declaration, but where is this handle defined?

📄 Start writing the definition of the `WEATHER_init()` function (`@TODO (1)` comment) by initializing the handle with the function arguments.

📄 In the `main()` function, find the most convenient place to call the Weather click initialization function. Compile and fix all errors and warnings.

It is no use uploading the firmware into the target MCU because the read and write functions have not been written yet.

### II.3.b. Weather click read function

✍ From the BME280 datasheet, find the information that explain how to read data from the sensor's registers, through its I<sup>2</sup>C interface. Write the steps down here.

✍ What is **your** slave address?

📄 Defined the `WEATHER_I2C_ADDR` macro-constant consequently.

✍ In the I<sup>2</sup>C peripheral HAL function, find and write down here the blocking transmit and receive functions.

📄 With these two HAL functions and the indications in the BME280's datasheet, complete the definition of the `WEATHER_readReg()` function in the `weather_click.c` file.

In order to test our readout function, let us read the `id` register value.

✍ What is the address of this `id` register? What is its purpose? What value should we read?

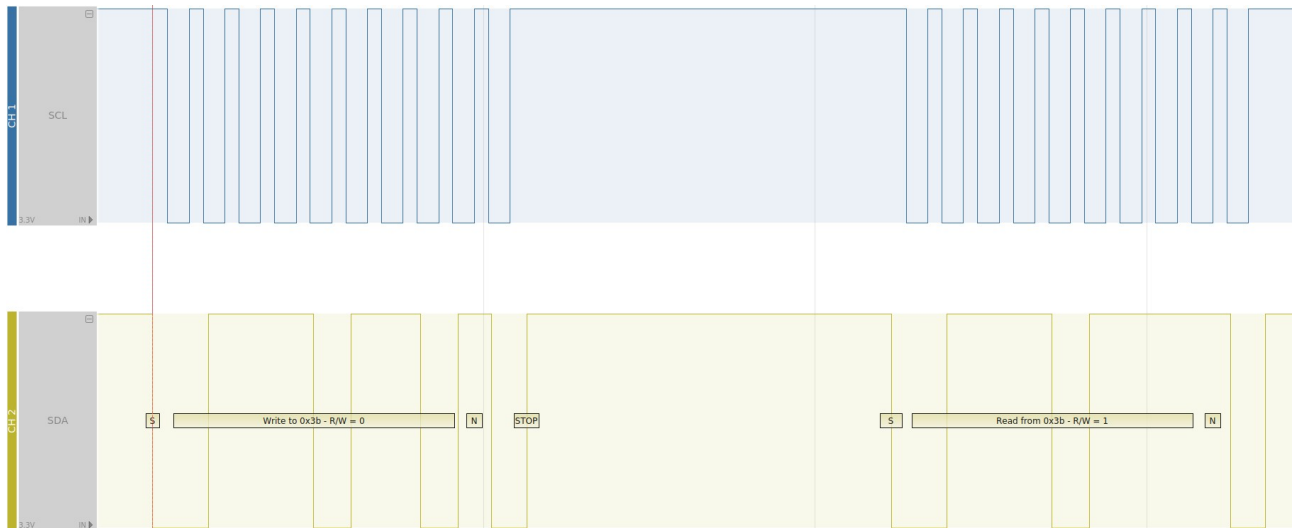
📄 Test the `WEATHER_readReg()` function in the `main()` by reading the `id` register's value.

📄 Send it to the serial terminal.

The value does not seem to match our expectations. If it does, it might your lucky day!



☞ Capture the exchanged messages with a logic analyzer. Check the frames and use an I<sup>2</sup>C protocol decoder. This is what you should observe.



✎ How many bytes should we observe, according to the datasheet's figure 10? Explain.

✎ What are the two bytes captured on the frames above?

✎ Which is the decoded address here? Is it the address you specified with your program?  
*(Note: the address bits (and data bits) are read on a SCL rising edge).*

✎ Compare these address bytes to the ones in the datasheet's figure 10?

✎ What is the meaning of the "N" or "ACK" bit indicated by the decoder? Is it normal?

✎ Get back to the HAL functions. Read the Doxygen comment block of the I<sup>2</sup>C transmit function `HAL_I2C_Master_Transmit()`. What is written about the `DevAddress` parameter?

✎ Same question for the `HAL_I2C_Master_Receive()` function.

Indeed the sensor's address is 7-bit long (*'The 7-bit device address is 111011x'*). But the I<sup>2</sup>C standard requires the address to take the seven most significant bits so that the least significant bit (bit 0) is used to indicate if this is a write operation (value '0') or a read operation (value '1').

📄 Change the `WEATHER_I2C_ADDR` macro-constant definition to use the most significant bits (in other words, let the LSb space empty).

🔧 Compile and try again to read the `id` register value and display it on a serial terminal.

🔧 Confirm that the value displayed on the serial terminal is the expected value.

🔧 Capture the exchanged frames with a logic analyzer.

🔧 Using the frame decoder and datasheet's figure 10, explain the role of each bit on the SDA line, specifying the bit producer (master or slave). Use as many details as possible.

What you should observe is that the SDA can be controlled by the master and the slave, but each of those has dedicated time slots (they cannot control the SDA line at the same time). This implies that the I<sup>2</sup>C protocol is **half-duplex**.

The default level of the SDA line is the high logic state, because of its pull-up resistor. More several devices can use the same SDA line because those ICs use open-drain outputs.

✎ How does a device write a logical '0'? And a logical '1'?

✎ Which is the logic state of an acknowledgment? Why is it not the opposite?

### II.3.c. Weather click write function

✎ From the BME280's datasheet, find the information for writing into the sensor's registers using an I<sup>2</sup>C interface. Write down here the different steps.

📖 With the I<sup>2</sup>C HAL functions and the datasheet's indications, complete the definition of the `WEATHER_writeReg()` function in `weather_click.c`.

We will test and confirm the write function later on. Firstly we will find which settings should be written into the BME280 so it matches our requirements. Only then we will write those values into the sensor's registers.

### II.3.d. BME280 configuration

Like numerous sensors, and like the LIS3DSH (accelerator) studied in the previous part, the BME280 has registers that contains adjustable parameters.

## 5. Global memory map and register description

### 5.1 General remarks

The entire communication with the device is performed by reading from and writing to registers. Registers have a width of 8 bits. There are several registers which are reserved; they should not be written to and no specific value is guaranteed when they are read. For details on the interface, consult chapter 6.

Table 18: Memory map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state	
hum_lsb	0xFE	hum_lsb<7:0>								0x00	
hum_msb	0xFD	hum_msb<7:0>								0x80	
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00	
temp_lsb	0xFB	temp_lsb<7:0>								0x00	
temp_msb	0xFA	temp_msb<7:0>								0x80	
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00	
press_lsb	0xF8	press_lsb<7:0>								0x00	
press_msb	0xF7	press_msb<7:0>								0x80	
config	0xF5	t_sb[2:0]			filter[2:0]			spi3w_en[0]		0x00	
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]		0x00	
status	0xF3	measuring[0]					im_update[0]				0x00
ctrl_hum	0xF2	osrs_h[2:0]								0x00	
calib26..calib41	0xE1...0xF0	calibration data								individual	
reset	0xE0	reset[7:0]								0x00	
id	0xD0	chip_id[7:0]								0x60	
calib00..calib25	0x88...0xA1	calibration data								individual	

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Chip ID	Reset
Type:	do not change	read only	read / write	read only	read only	read only	write only

✎ Find in the datasheet the three control registers to configure.

✎ In one of those registers, the `mode[1:0]` field must be set. Write the three operation modes of the sensor. Which one allow us to perform a periodic readout of the physical quantities?

✎ For the three control registers, find the bits values following these requirements:

- measure all three physical quantities, with no oversampling
- read out every 500 ms
- no filter

Now that the control parameters have been determined, let us switch now to the device's configuration by completing the definition of the `WEATHER_init()` function.

📖 The first thing to do when trying to configure a device is to check if it is reachable. Complete the sensor's identification test following the `@TODO (2)` comment (you can delete the equivalent code, now redundant, in the `main()` function).

📖 Configure the BME280 by writing the values determined above into the control registers. Note that the write order has an impact (see datasheet).

📖 Last read the sensor's calibration parameters<sup>13</sup> by calling the `WEATHER_getCalibParams()` function after the `@TODO (4)` comment. This function is provided with its complete definition.

---

**13** Calibration parameters will be explained in " Part 4 - II.3.e ".

The initialization function is now written. We will test and confirm it. This will be the occasion to try the operation of the `WEATHER_writeReg()` function too.

☞ In the `main()`, call the `WEATHER_init()` function at the application's start.

☞ In the main loop of the application, write instructions that read the value of the configured registers.

☞ Display their values onto the serial terminal and compare them to the expected values.

☞ Once that the initialization and write function have been confirmed, use a logic analyzer to capture a write operation into a BME280's register.

☞ Using the decoding tool and datasheet's figure 9, explain the role of each bit on the SDA line, and specify which node (master or slave) wrote each bit. Be as accurate as possible.

Do not pass this point until this page has been checked by the teacher.

### II.3.e. Measurements readout

In order to understand the various settings that can impact the temperature, pressure and data readouts, we need to carefully read the BME280's datasheet. Yes, this device is more complex than it seems. The following datasheet sections must be read to answer the upcoming questions:

- 3.4 Measurement flow (3.4.3 included)
- 4. Data readout
- 5.4 Register description (for the registers you will use)

✎ What is the point of oversampling acquisitions?

✎ What is the point of using a filter (here and infinite impulse response filter)?

✎ What is the resolution of the humidity, pressure and temperature values in our case (we want no filter, no oversampling)?

✎ Which registers holds the results of all three measurements?

✎ Does the number of bits in these registers match with the resolution in our case? What should we do if there are more bits in the registers than in the expected resolutions?

✎ Let us take the temperature measurement as an example. After reading the temperature value out of the registers, how do you build a temperature value in a integer-type C variable?

The integer value is actually not a true temperature value. It is written in the datasheet that each sensing element is different (i.e. from a device to another). As a consequence the calibration parameters of each BME280 have been determined during the manufacturing process and stocked into the device's NVM (Non-Volatile Memory). These parameters, accessible through some registers, will be used to "compensate" the measured values in order to calculate the true values.

The datasheet gives example compensation codes for the three quantities (temperature, pressure, humidity) in section "4.2.3 Compensation formulas". In these code extracts, the values are stored on 32-bit integer variable using a **fixed-point arithmetic**. Bosch Sensortec also provides a **floating-point arithmetic** version of those codes on its GitHub repository<sup>14</sup>.

In the `weather_click.c` file, the given codes have been rewritten to match the driver's coding style (function prototypes, values, structures, ...). We chose to use the fixed-point arithmetic version rather than the floating-point arithmetic one.

✎ Why do we have to make the operations on fixed-point variables and not on floating-point? *Hint: this is due to our MCU<sup>15</sup>.*

✎ What is a fixed-point number? What are the advantages of this notation? Have a look on the Internet or ask the teacher.

<sup>14</sup> [https://github.com/boschsensortec/BME280\\_driver/tree/master](https://github.com/boschsensortec/BME280_driver/tree/master)

<sup>15</sup> This questions is sometimes wrong, depending on the used STM32...

Take a look into the Doxygen comment blocks of the three compensation function, in the `weather_click.c` file. The interesting point here is the format and the unit of the returned values, i.e. `fine_temp`, `fine_press` and `fine_hum`<sup>16</sup>.

✍ Write a pseudo-code of the conversion of the temperature value `fine_temp` (32-bit integer format, in 0.01 °C) into a `temp_degC` in floating-point format, in °C.

✍ Same question for converting `fine_hum` (Q22.10 fixed-point integer format, % RH) into `hum_rh` (floating-point format, % RH) (% RH = percentage of relative humidity).

✍ Same question for `fine_press` (Q24.8 integer format, in Pa), to `press_hPa` (floating-point format, in hPa).

With the answers to the previous questions it is now possible to complete the definition of the read functions for all three quantities.

📄 Complete the definition of the `WEATHER_getTemperature()`, `WEATHER_getPressure()` and `WEATHER_getHumidity()` functions, following your previous answers.

📄 Complete the definition of the `WEATHER_getAll()` function, by simply calling the three functions.

📄 Test this functions in the `main()` using the following code lines:

```
float temp, press, hum;
WEATHER_getAll(&press, &temp, &hum);
sprintf( uart_out, "WEATHER\t temperature = %f degC\n\r", temp );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
sprintf( uart_out, "WEATHER\t pressure = %f hPa\n\r", press );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
sprintf( uart_out, "WEATHER\t humidity = %f %%RH\n\r", hum );
HAL_UART_Transmit( &huart2, (uint8_t*)uart_out, strlen(uart_out), HAL_MAX_DELAY );
```

📄 Confirm with a serial terminal.

<sup>16</sup> These information come from the datasheet, section '4.2.3 Compensation formulas'.



### II.4. Overview

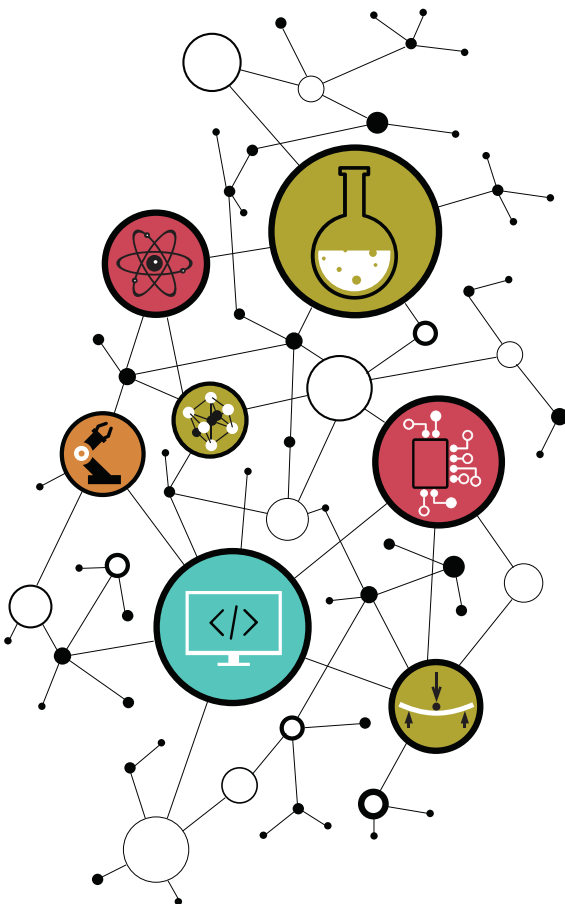
✍ Go through all points of this part, and write here a summary of what you understood. This will be available for you if needed for future developments.

1. About the physical layer (wires, signals, voltage levels, logic levels, ...)
2. About the protocol layer (number of bits, who owns each line, ...)
3. About the firmware layer (peripheral configuration, used functions)
4. About the sensor layer (parameters, registers access, ...).



PART 5

# GLOSSARY



LSb / MSb	<i>Least/Most Significant bit</i>
LSB / MSB	<i>Least/Most Significant Byte</i>
<b>MCU</b>	<b><i>Microcontroller Unit</i></b>
CPU	<i>Central Processing Unit</i>
Peripheral	Internal hardware component of the MCU, designed for a specific task
GPIO	<i>General Purpose Input/Output</i> , peripheral that controls the MCU pins
<b>STM</b>	<b><i>STMicroelectronics</i></b> , French-Italian semiconductor company
STM32	32-bit MCU series from ST, built around ARM Cortex-M cores
STM32CubeIDE	Integrated Development Environment for ST processors
STM32CubeMX	Graphical initialization code generator for ST processors
HAL	<i>Hardware Abstraction Layer</i> , functions for an easy use of MCU peripherals
STLink VCP	<i>STLink Virtual Com Port</i> , accessible through the USB link of the NUCLEO
NVIC	<i>Nested Vector Interrupt Controller</i> , ARM CPUs interrupt controller
IF	<i>Interrupt Flag</i> , indicates that an event has occurred
IRQ	<i>Interrupt Request</i> , asks the CPU to pause its execution
ISR	<i>Interrupt Service Routine</i> , function executed when an IRQ stops the CPU
Callback	Function called by an ISR and which definition must be written by the dev
<b>UART</b>	<b><i>Universal Asynchronous Receiver-Transmitter</i></b> , serial comm peripheral
USART	<i>Universal Synchronous-Asynchronous Receiver-Transmitter</i>
Tx	<i>Transmit line</i>
Rx	<i>Receive line</i>
<b>SPI</b>	<b><i>Serial Peripheral Interface</i></b>
	Bidirectional, full-duplex, synchronous serial bus, designed by Motorola
MISO	<i>Master Main In, Slave Sub Out</i>
MOSI	<i>Master Main Out, Slave Sub In</i>
$\overline{SS}$	<i>Slave Select</i> , active low
$\overline{CS}$	<i>Chip Select</i> , active low
<b>I<sup>2</sup>C</b>	<b><i>Inter-Integrated Circuit</i></b>
	Bidirectional, half-duplex, synchronous serial bus, designed by Philips
SDA	<i>Serial Data Line</i>
SCL	<i>Serial Clock Line</i>
ACK	<i>Acknowledgment</i>
NACK	<i>Non-Acknowledgment</i>





