SERIAL COMMUNICATION BUSES

Lab materials

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

GPSE / IPC-SATE-ECSE
dimitri.boudier@ensicaen.fr

Contact

Dimitri Boudier – In charge of this course
dimitri.boudier@ensicaen.fr

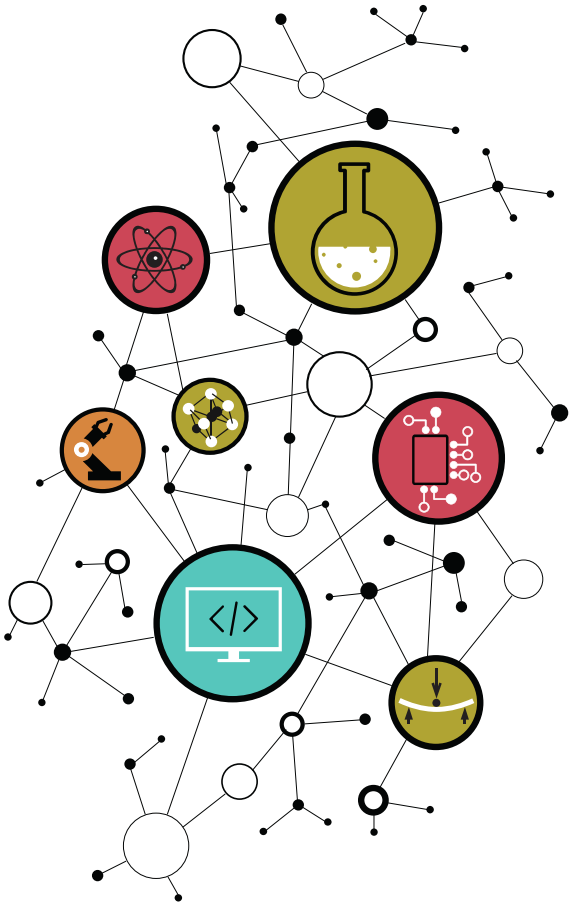
Resources

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

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 **I²C**. 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.

I.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.

I.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.

I.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 et I²C)
  - *Note: this is the only optional equipment of this list (but still very useful)*

The software are the following:

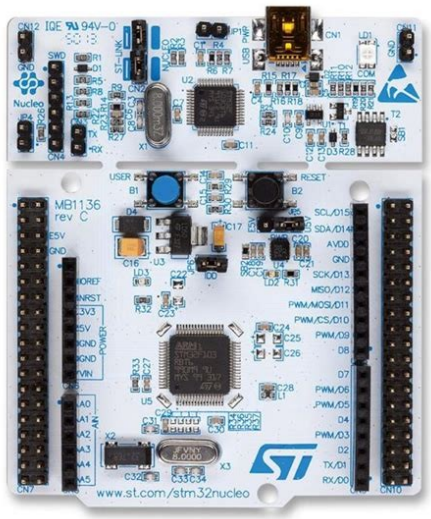
- STM32CubeIDE (validated with version 1.11.0) :
  - IDE used for programming and debugging the STM32 NUCLEO boards
  - https://www.st.com/en/development-tools/stm32cubeide.html
  - any other compatible IDE (Keil, VS Code with the STM32 extension, …) is accepted
  - *Note: there is a tutorial in this 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 STMicroelectonics 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**[1] board used in this document has on its PCB the target MCU (STM32L073RZ), a programming/debugging probe, a push-button and a LED.

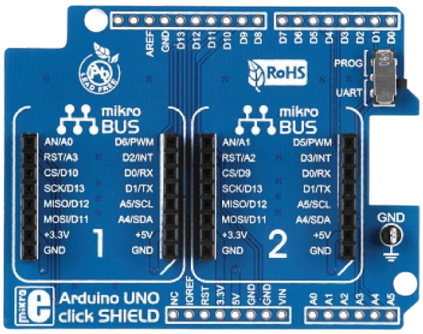
The NUCLEO boards also come with two connectors that give an easy access to the MCU pins. This facilitates the prototyping process. You can see on this board an Arduino connector and a Morpho connector, the latter being specific to STM (2 x 38 broches).

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 debuggind interface, with no additional equipment needed.

For more information please see the dedicated tutorial document: `tutorials/tutoriel_nucleol073rz`.

The processor embedded onto the NUCLEO-L073RZ board is a STMicroelectonics **STM32L073RZ**[2] 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 I²C: 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.

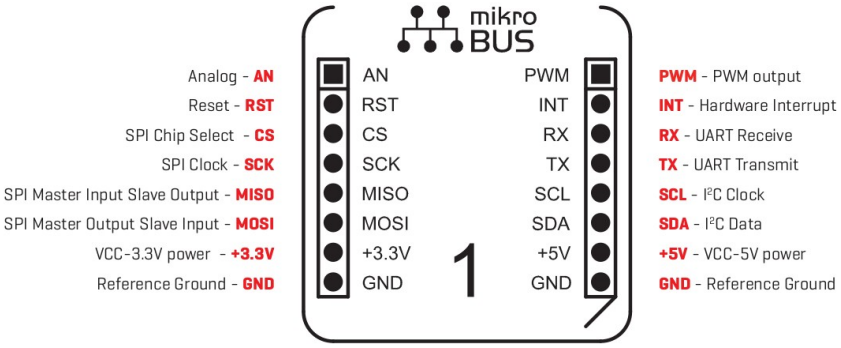


1 https://www.st.com/en/evaluation-tools/nucleo-l073rz.html
2 https://www.st.com/en/microcontrollers-microprocessors/stm32l073rz.html

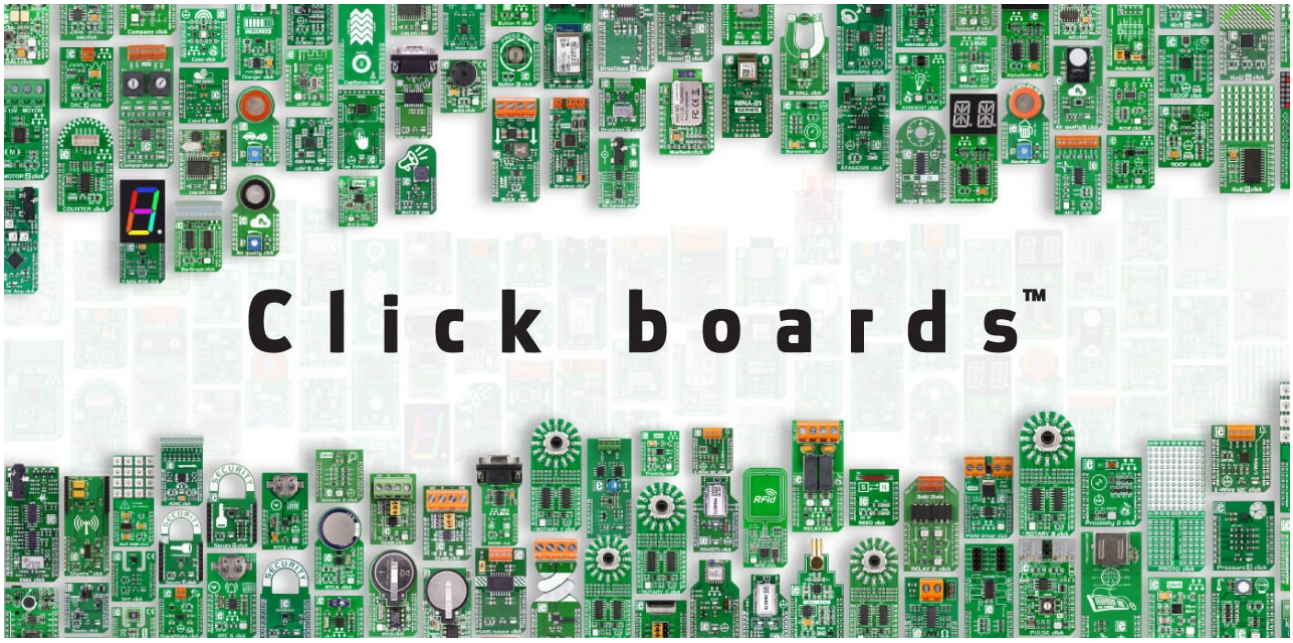
II.2.b. MIKROE Click board™

MikroElectronika[3] (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, I²C, analogue, PWM) makes the mikroBUS™ compatible with a wide set of different uses.



The mikroBUS™ standard helped MIKROE to spread is 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 septembre 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).



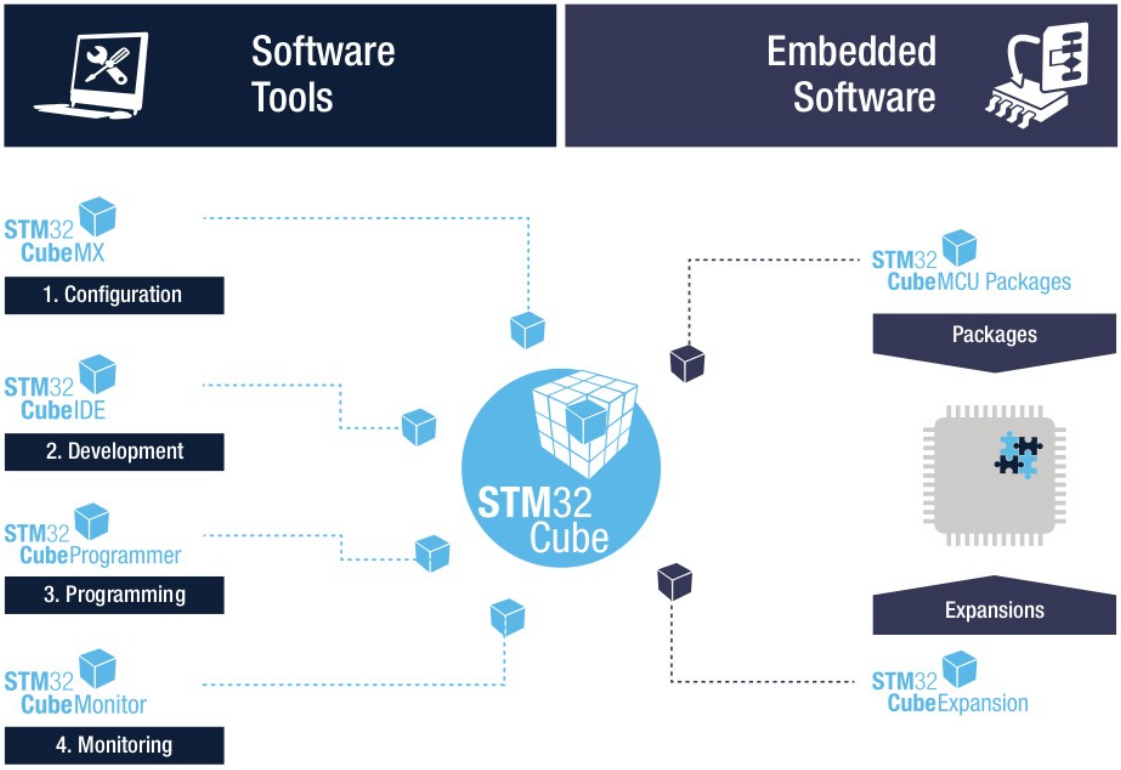
For each Click board, MIKROE gives access to the board schematic, to the ICs' technical documentation and to the drivers they have developed. Everything is available on the manufacturer website. For the sensors used in this lab, all the necessary files have bee gathered in the `connectivity/datasheets/` 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 STMicroelectonics: **STM32Cube**[4]. It is a 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 `tutoriel_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.

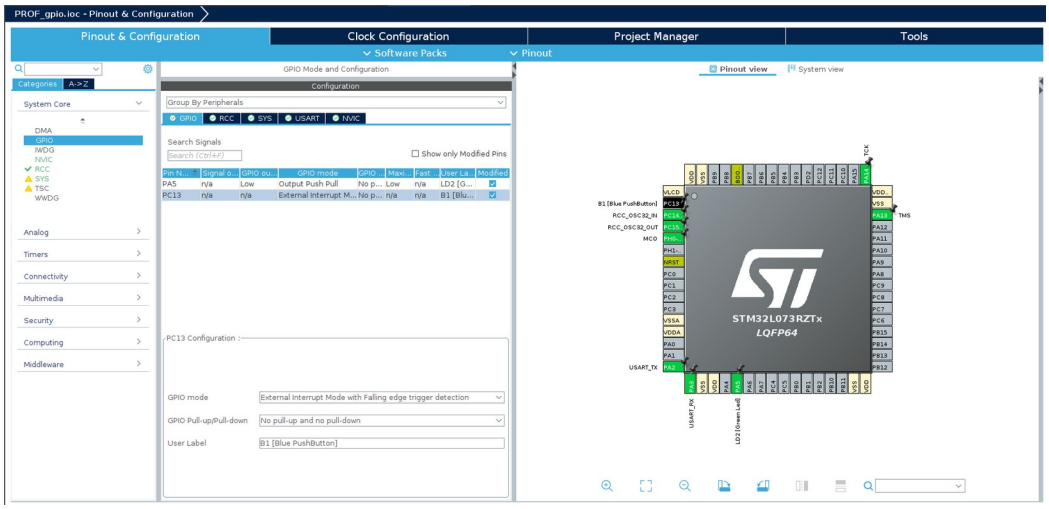
III.1. Creating a project

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 informations 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.



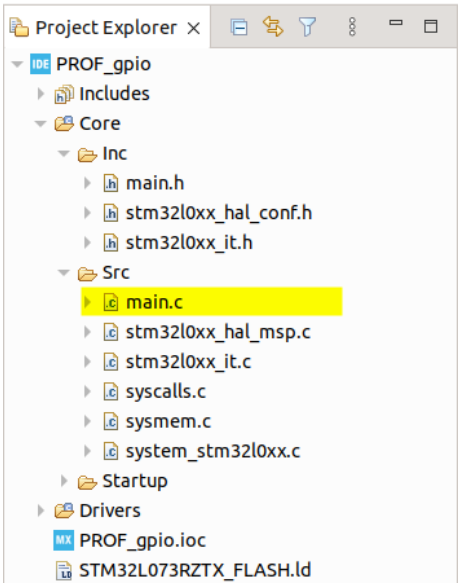
In `System Core` → `GPIO`, note here the pin number of the configured GPIOs, and their function as well:

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 (CuveMX), 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 is 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+clic` 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+Clic` on the `HAL_GPIO_Init()` funciton call. This time another file opens: `stm32l0xx_hal_gpio.c`. This file belongs the 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 `tutoriel_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 theses constants definitions? Who/what defined them here?

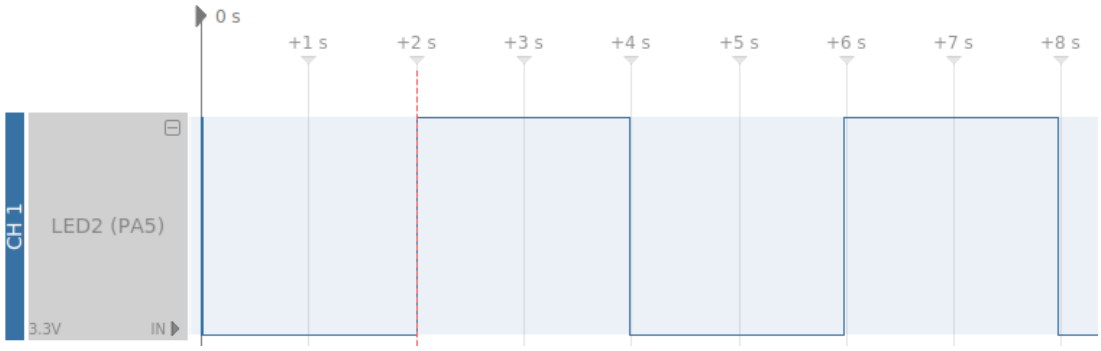
✎ Which function let us change the GPIO logic state? Take a look among the numerous functions declared in the `stm32l0xx_hal_gpio.h` file.

✎ 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.

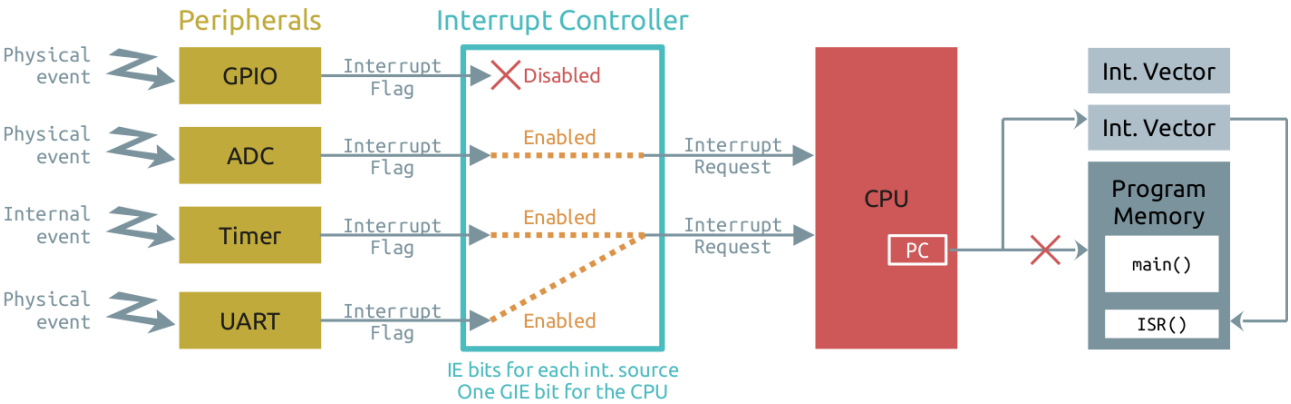


Example of what you should observe with a logic analyzer.

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 theses 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.

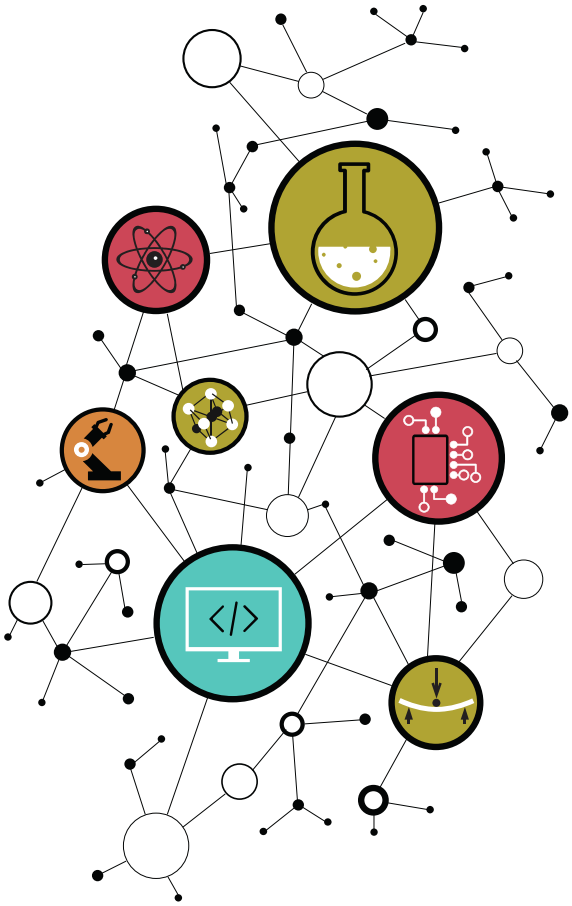
🗗 Confirm with an oscilloscope or a logic analyzer, triggered on the falling edge of the push button pin.

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

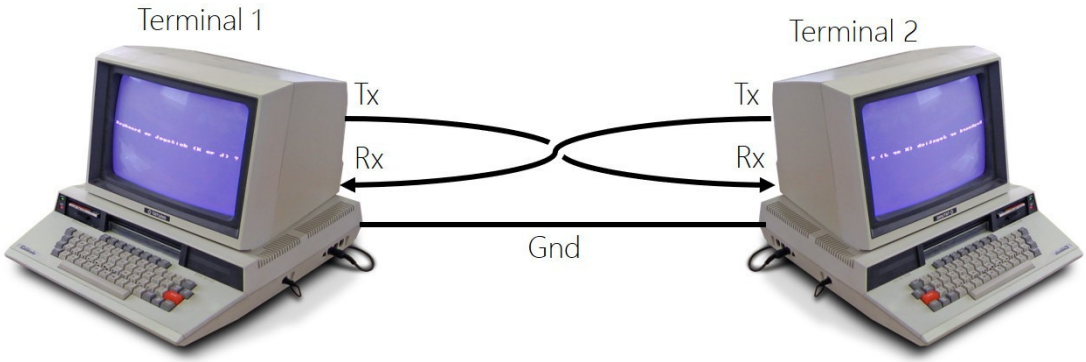


I. Presentation

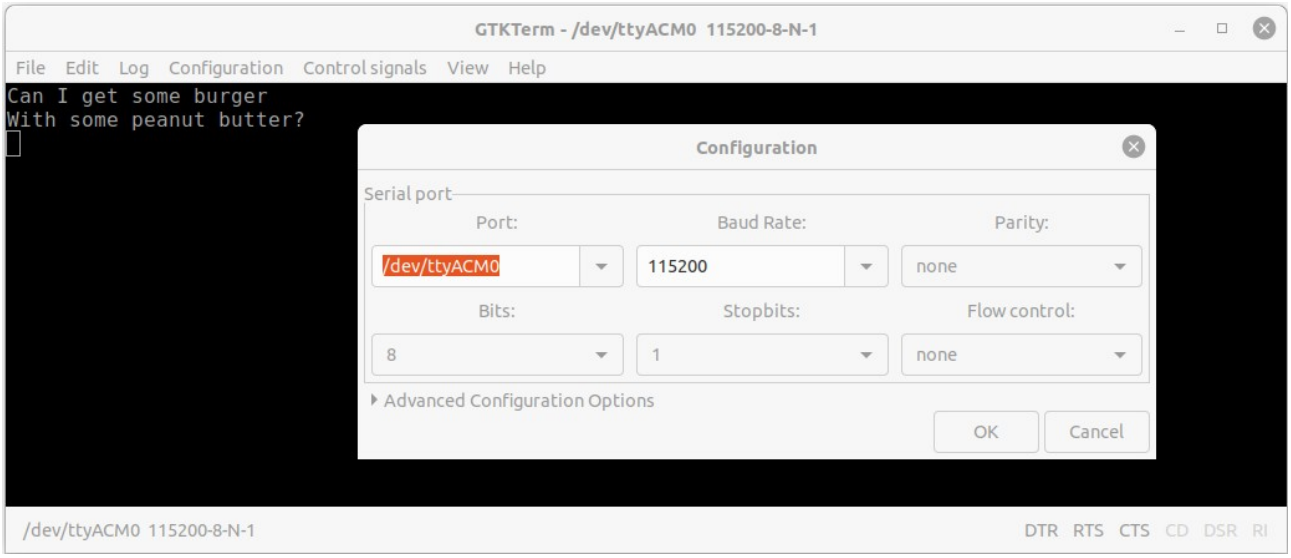
The **UART** is an ***Universal Asynchronous Receiver-Transmitter***. It names a device or a peripheral but it is not a communication protocol, whatever the misuse of language frequently seen. The communication between two UARTs is an **asynchronous serial communication**.

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

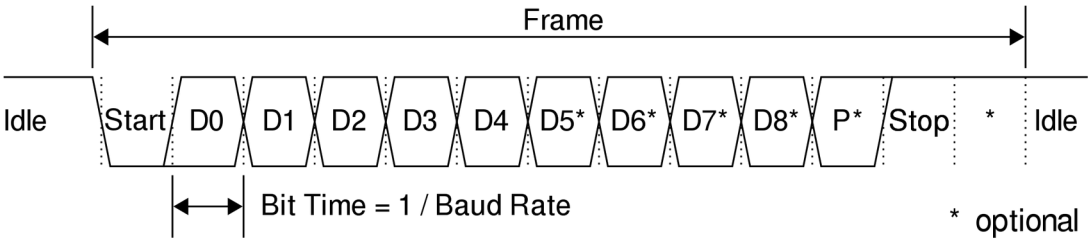
From a hardware view, a UART peripheral contains two lines:

- 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[5] 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 *LS**b**/MS**b*** notation for *Least/Most Significant **b**it*, and the *LS**B**/MS**B*** notation for *Least/Most Significant **B**yte*.

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+Clic` 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 1st 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.

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

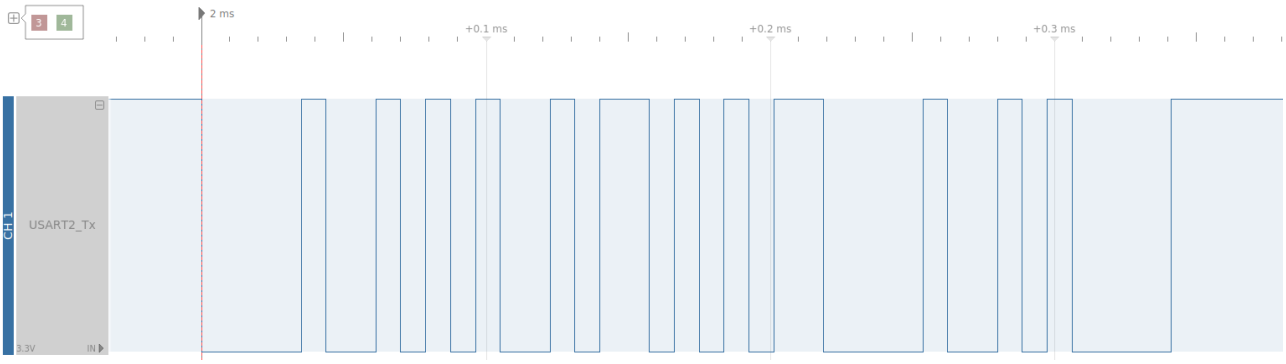
Now it is time to analyze the frame using a powerful tool: the logic analyzer. To do so you will have to probe the Rx and Tx electric signals. But we have seen that these pins are not linked to the NUCLEO connectors, but to the STLink. If we want to analyze those signals, we will have to find a way to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

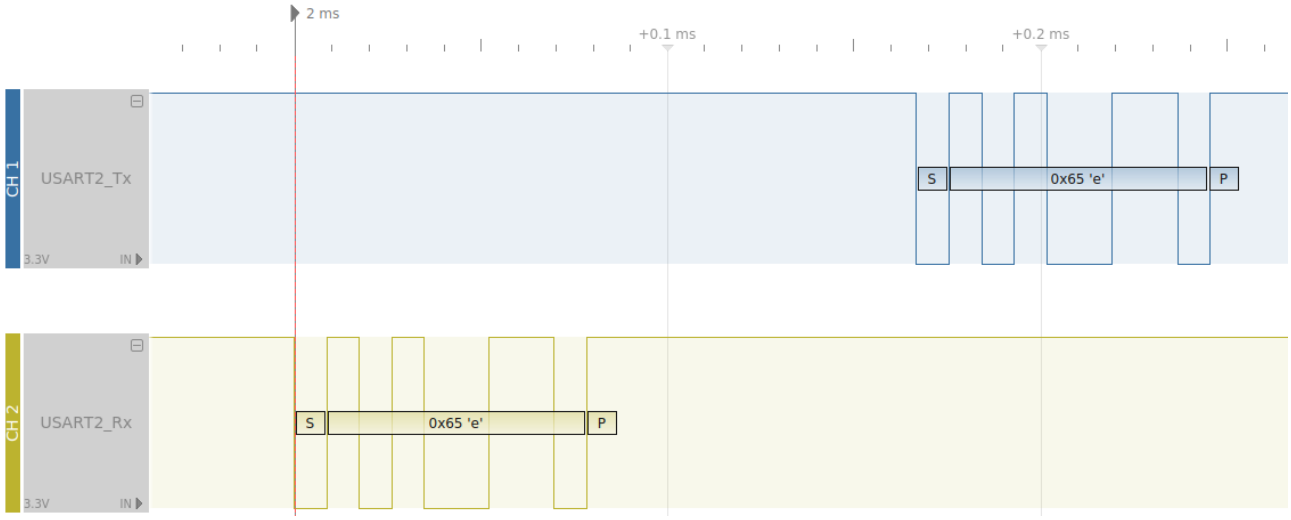
✎ What arguments should you pass to the `HAL_UART_Receive()` function?

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 and 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 use polling. In other words it is a blocking function: it monopolize 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 1III.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 ISR 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 follow 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 functions ends and the code executions 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[6]. 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?

6 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.

Example of what you could observe on a serial terminal.

III.5. Bridge UART (pour les plus avancés)

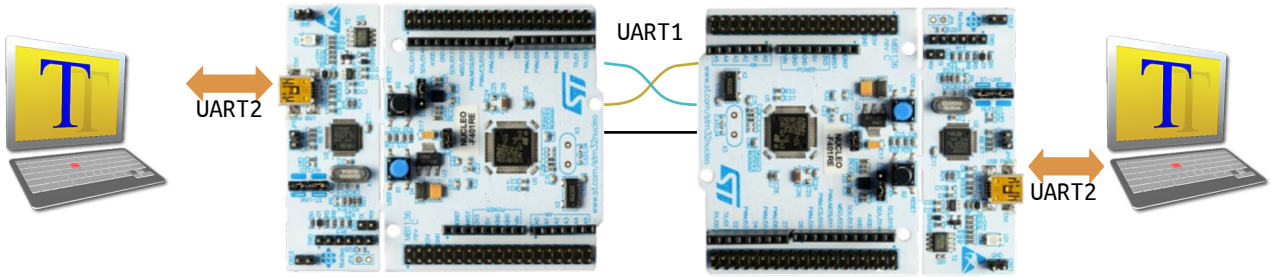
Vous allez dans cette partie créer un bridge UART. Dans le langage des bus de communication, un bridge est une interface qui ne fait que retransmettre un message venant d'une interface vers une autre, avec généralement une conversion de protocole entre les interfaces. Autrement dit, cela change la forme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

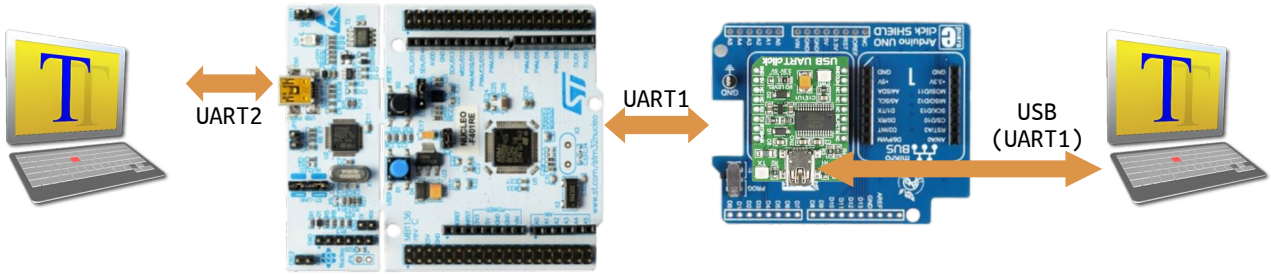
D'un point de vue matériel, vous avez le choix entre deux solutions.

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.

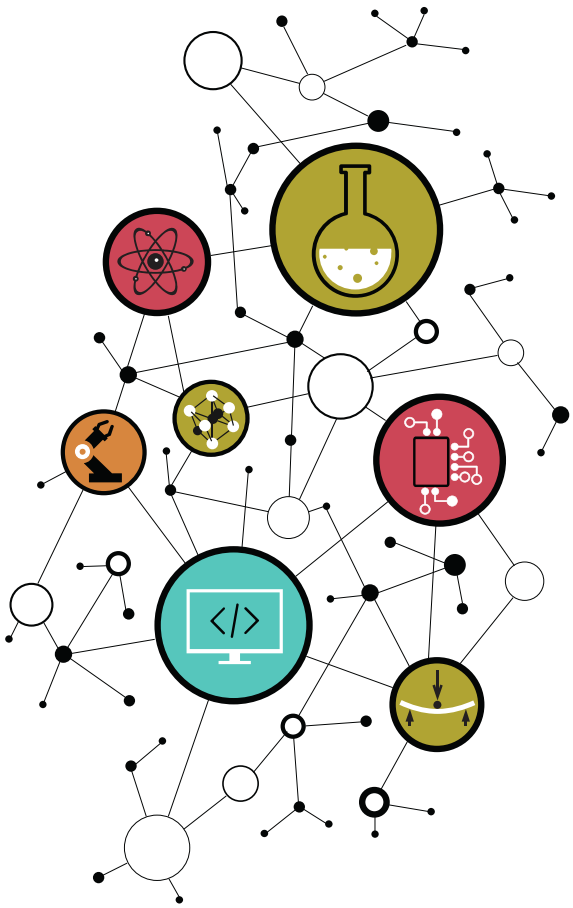
IV. Overview

✎ 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[7]. 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*.

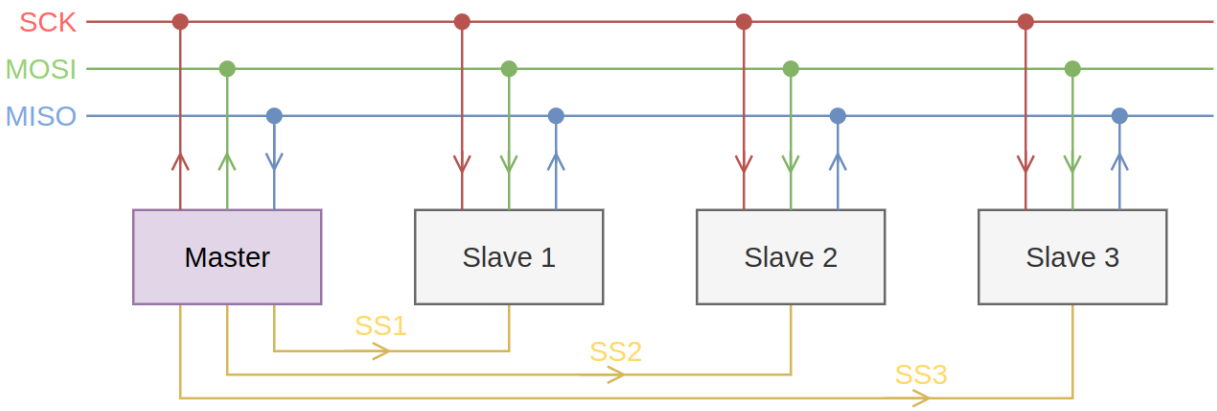
It has been decided for this course to use the original terminology, because (1) every electronic engineer knows these terms (whether they use it or not), because (2) not all online resources (webistes, datasheets, tutorials, …) have evolved (and if they did, they did not chose the same terms) and because (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:

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



7 https://en.wikipedia.org/wiki/Master/slave_(technology)

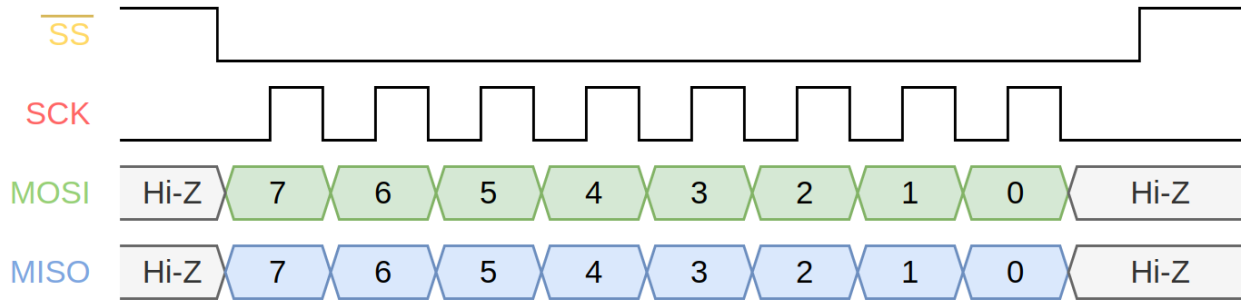
II. Frame constitution

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 $\overline{SS}$ 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 stabilise).



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 »[8].

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.

8 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

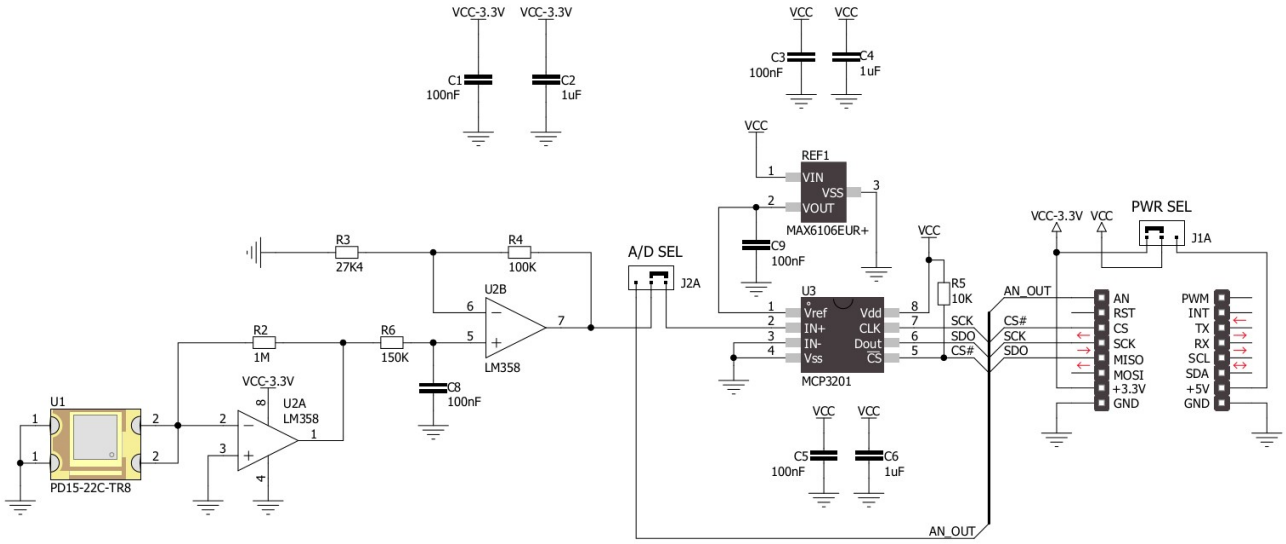
III. Application: Light click

The first device you will communicate with is a Click board™ from the MIKROE company: the Light click[9]. 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 (shematics, 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 whih 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[10] 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?? 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 S*TM32L0x3 user manual* counts 50 pages (for a total of 1,040). The peripheral itself uses nine registers.

10 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*)"\r\nApp 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[11]? 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 Il est en réalité possible de configurer ce périphérique SPI pour utiliser des trames de 4 à 16 bits. Nous gardons une taille de 8 bits par soucis de généricité (et de compatibilité avec les prochains 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 = %6.2f%%\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!

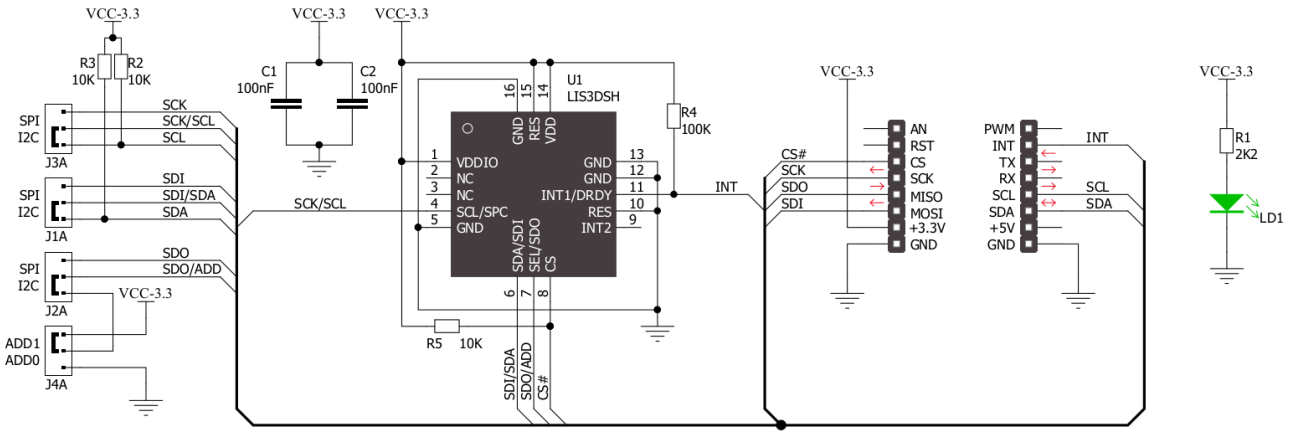
IV. Application: Accel 2 click

Nous allons maintenant utiliser un autre Click board™ : le Accel 2 click[12]. Comme vous vous en doutez il s'agit d'un accéléromètre. Nous communiquerons toujours en SPI, mais vous allez voir que ce composant est un peu plus complexe que le Light click.

IV.1. Étude du capteur

Ici l'étude du schéma n'a que peu d'intérêt puisque le Accel 2 click ne contient qu'un seul circuit intégré (IC), l'accéléromètre 3-axes LIS3DSH de STMicroelectronics. Nous pouvons toutefois noter la présence de quatre cavaliers (*jumpers*) sur la gauche du schéma.

✎ À quels composants ces *jumpers* correspondent-ils physiquement sur la carte ? Quelle est leur fonction ?



✎ En plaçant le Accel 2 click sur un emplacement libre du shield, indiquez à quelles broches du MCU sont reliés les signaux MOSI, MISO, SCK et CS.

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

IV.2. Configuration du périphérique SPI

Vous avez pu constater que le périphérique SPI n'a pas besoin d'être reconfiguré puisqu'il utilise les mêmes broches que le Light Click : c'est d'ailleurs l'intérêt d'un bus de communication.

Les broches MOSI, MISO et SCK sont ici communes au maître et à tous les esclaves.

En revanche, le signal CS étant unique pour chaque esclave, il reste à configurer la GPIO qui pilotera le signal CS de l'Accel 2 click.

💻 Reprenez le projet STM32CubeIDE `VOTRENOM_spi`.

💻 Ouvrez le fichier `VOTRENOM_spi.ioc` afin de configurer la GPIO correspondant à CS.

- System Core → GPIO
  - Sur la vue composant : clic gauche sur la broche → `GPIO_Output`
  - Sur le volet « GPIO Mode and Configuration », P?? configuration :
    - GPIO Output level = High
    - …
    - User Label = ACCEL_2_nSS

💻 Générez le code ( ).

Vous pourrez constater que la fonction `MX_GPIO_Init()` du `main.c` contient désormais l'initialisation de la GPIO.

IV.3. Driver accel_2_click

Abordons maintenant le driver de l'Accel 2 click.

💻 Les fichiers drivers se trouvent dans le répertoire `connectivity/workspace/drivers/`. Copiez-collez les fichiers `accel_2_click.c` et `accel_2_click.h` dans les répertoires du projet `…/CM4/Core/Src/` et `…/CM4/Core/Inc/` respectivement. Depuis l'explorateur de projet de l'IDE, vous allez sûrement devoir rafraîchir ces deux dossiers (F5) pour voir apparaître les nouveaux fichiers.

À première vue, le *header* est bien plus complet que celui du Light click. Mais à y regarder de plus près, la première partie ne contient que des macro-constantes (nous y reviendrons). Quant à ce qui suit, cela ressemble d'assez près au driver précédent : on retrouve une structure (`ACCEL_2_handle_t`, qui contient les informations de l'Accel 2 click après initialisation), une fonction d'initialisation, quelques fonctions de lecture et une fonction d'écriture.

```
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);
```

Dans le fichier C, les définitions des fonctions restent à compléter. Nous les aborderons étape par étape.

IV.3.a. Initialisation du handle

🔪 Parcourez les fichiers `main.c` et `main.h` pour retrouver le handle du périphérique SPI et la définition du port et de numéro de la broche SS de l'Accel 2 click. Comment se nomment-ils ?

💻 Débutez la définition de la fonction `ACCEL_2_init()`, (commentaire `@TODO (1)`) en initialisant le *handle* avec les paramètres fournis à la fonction.

💻 Dans la fonction `main()`, trouvez l'endroit le plus approprié pour appeler la fonction d'initialisation de l'Accel 2 click. Compilez et éliminez toute source d'erreur ou de warning.

Il n'y a pas lieu d'exécuter le programme sur cible puisque les fonctions de lecture et d'écriture n'ont pas encore été rédigées.

IV.3.b. Fonction de lecture depuis le périphérique

✎ D'après le chapitre « 5 Digital interfaces » de la datasheet du LIS3DSH, trois modes de communication sont disponibles. Quels sont-ils ?

✎ La table 8 de la datasheet montre que la dénomination des broches est différente de ce qu'on voit habituellement en SPI (CS, SCK, MOSI, MISO). En effet, bien que standardisés, les noms des broches changent parfois selon le fondeur. Ne pas hésiter à noter ici la traduction si besoin.

✎ Comme toute documentation qui se respecte, celle-ci explique comment accéder à une donnée stockée dans le composant. Retrouvez ces informations et retranscrivez-les ici.

💻 À partir de votre précédente réponse, complétez la définition de `ACCEL_2_readReg()`. Aidez-vous du cartouche Doxygen qui précède la déclaration de cette fonction (dans le `.h`).

💻 Dans le `while(1)` du `main()`, ajoutez les lignes suivantes. Compilez et ajustez le code de sorte à corriger les erreurs et 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 );
```

✎ Que fait cette portion de code ? Quelle valeur est censée s'afficher sur le terminal série ?

🗗 Confirmez avec un terminal série.

🗗 Effectuez une capture de cet échange avec un analyseur logique. Comparez la capture avec la « Figure 8. SPI read protocol » de la datasheet. Vous devriez constater des différences. Expliquons-les grâce aux questions suivantes.

✎ D'après la datasheet, quel devrait être l'état de repos de la ligne SCLK ? Qu'observez-vous sur votre capture ?

✎ D'après la datasheet, à quel instant les lignes MOSI et MISO devraient lire chaque bit ? Qu'observez-vous sur votre capture ?

✎ D'après la datasheet, à quel instant les lignes MOSI et MISO devraient écrire chaque bit ? Qu'observez-vous sur votre capture ?

Ces informations correspondent aux **modes (CPOL, CPHA)**. Le paramètre CPOL (*Clock POLarity*) indique le niveau de repos de la ligne d'horloge ('0' = niveau bas, '1' = niveau haut). Le paramètre CPHA (*Clock PHAse*) indique si la donnée (sur MOSI *et* MISO) est échantillonnée au premier front (valeur '0') ou au second front ('1') d'horloge.

Ainsi, les modes (0, 0) et (1, 1) sont compatibles en ce qui concerne l'échange de données puisque dans les deux cas les bits sont produits sur front descendant d'horloge et sont lus sur front montant. Seul l'état de repos du signal d'horloge change. Similairement, les modes (0, 1) et (1, 0) sont compatibles entre eux.

✎ À quel mode correspond la section « 5.2 SPI bus interface » de la datasheet du LIS3DSH ?

✎ Dans quel mode a été configuré notre périphérique SPI1 ? Réponse dans le fichier `.ioc`.

Conclusion : configuré ainsi, le périphérique SPI de notre MCU est bien compatible avec le LIS3DSH, même si ce n'est pas exactement le mode demandé par l'accéléromètre.

D'ailleurs si on revient à l'ADC du Light click, la datasheet du MCP3201 indique : « *SPI Mode 0,0 (clock idles low) and SPI Mode 1,1 (clock idles high) are both compatible with the MCP3201* », comme en témoignent les figures 6-1 et 6-2 présentant ces deux modes.

IV.3.c. Fonction d'écriture vers le périphérique

Le LIS3DSH contient de nombreux registres, dont certains contiennent les paramètres de configuration du capteur. Nous aurons donc à modifier ceux-ci afin d'opérer dans le mode de fonctionnement qui nous intéresse.

✎ Retrouvez dans la documentation les informations indiquant comment modifier une donnée stockée dans le composant, et retranscrivez-les ici.

- Complétez la définition de la fonction `ACCEL_2_writeReg()`.
- Validez en ajoutant le code suivant au `while(1)` du `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 );
```

✎ Quelles sont les opérations réalisées par ces lignes ? *(Note : il n'est pas demandé de déterminer le rôle du registre VFC_1 : il ne sera pas utilisé par la suite et sa valeur n'a pas d'impact sur le fonctionnement du capteur dans notre étude).*

- Confirmez sur terminal série que votre fonction `ACCEL_2_writeReg()` est fonctionnelle.
- Relevez avec analyseur logique une demande d'écriture et la lecture correspondante.

IV.3.d. Configuration du LIS3DSH

Maintenant que les fonctions de communication SPI entre le MCU et l'Accel 2 click sont opérationnelles, nous allons pouvoir configurer le LIS3DSH de sorte à répondre à nos besoins. C'est ici la principale différence avec le précédent capteur (le Light click) qui n'était pas configurable.

Observez le chapitre « 6 Register mapping » de la *datasheet*. Vous constaterez que tous les registres sont listés ici, avec leur adresse, leur mode d'accès (r, w, r/w), leur description et éventuellement leur valeur par défaut (i.e. à la mise sous tension). Vous avez peut-être remarqué que ces registres ont été listés dans le fichier `accel_2_click.h` à l'aide de macro-constantes qui imposent leur adresse, de sorte à pouvoir les utiliser depuis le driver.

Quand il découvre la datasheet d'un nouveau capteur, le développeur expérimenté cherchera toujours les registres nommés `CTRL`, `CFG`, `SETTINGS`, ou autre nom similaire. Ceux-ci permettent de cibler rapidement les paramètres réglables du composant. Avec quelques allers-retours dans la *datasheet*, cela permet d'avoir une configuration qui fait le strict minimum. Dans un second temps, une lecture approfondie de la documentation est nécessaire pour configurer un composant de sorte à répondre précisément au cahier des charges. Toutefois savoir paramétrer un capteur n'est pas l'objectif principal de ce module. Aussi le travail de ciblage des registres et des paramètres est préparé par les questions suivantes.

✎ En vous mettant dans la peau d'un développeur aguerri, trouvez les registres qui devraient permettent de configurer le LIS3DSH (il y en a 6 ou 7).

En parcourant la description de tous ces registres, une première lecture permet d'extraire les bits intéressants (en surligné, les mots-clés que cherche le développeur expérimenté) :

ODR3	ODR2	ODR1	ODR0	BDU	Zen	Yen	Xen

[redacted] register description

ODR 3:0	Output data rate and power mode selection. Default value: 0000 (see *Table 55*)
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 enable. Default value: 1 (0: Z-axis disabled; 1: Z-axis enabled)
Yen	Y-axis enable. Default value: 1 (0: Y-axis disabled; 1: Y-axis enabled)
Xen	X-axis enable. 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.

✎ Retrouvez le registre correspondant et justifiez la valeur que doivent prendre ces bits.

Avant de réellement configurer un composant, il convient de s'assurer que celui-ci est effectivement présent sur le bus. C'est là le rôle du registre `WHO_AM_I` qui contient une valeur connue à l'avance : il suffit de lire ce registre et comparer la valeur lue à celle attendue pour confirmer (ou infirmer) la présence du capteur.

💻 Complétez la définition de la fonction `ACCEL_2_init()` en suivant les commentaires `@TODO (2)` (vérification de la présence du capteur) et `@TODO (3)` (configuration du capteur).

💻 Compilez, téléversez.

🗗 Validez avec le terminal série que le programme se bloque si le capteur n'est pas présent au démarrage de l'application.

IV.3.e. Lecture des valeurs d'accélération

Nous allons nous charger de récupérer la valeur d'accélération sur l'axe z, en définissant la fonction `ACCEL_2_readZ()`. Il restera ensuite à répliquer le code pour les autres fonctions.

💻 Mais tout d'abord, ajoutez le code suivant au `while(1)` du `main()`. Celui-ci nous permettra de tester régulièrement la fonction à développer (même si elle n'est pas encore fonctionnelle).

```
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 );
```

🗗 Compilez, téléversez et observez le résultat sur terminal série. Le corps de la fonction `ACCEL_2_readZ()` étant vide, la valeur d'accélération affichée est toujours `0.000000`, mais ce code de test est prêt à être utilisé avec les développements qui arrivent.

Justement, passons au développement de la fonction `ACCEL_2_readZ()`.

💻 `@TODO (1)` : Avec la datasheet, déterminez le(s) registre(s) à lire pour récupérer la valeur d'accélération sur l'axe z.

💻 `@TODO (2)` : À partir des données récupérées, construisez un nombre à virgule fixe. Il peut être judicieux de passer par une variable intermédiaire de type entier sur 16 bits.

🗗 Compilez, téléversez et observez le résultat sur terminal série. Cette fois-ci la valeur d'accélération devrait évoluer, notamment en bougeant la carte.

Certes la valeur affichée évolue avec l'orientation de la carte, mais son ordre de grandeur ne correspond à unité ni en $m \cdot s^{-2}$, ni en g. La documentation du capteur nous donne les informations suivantes :

Table 3. Mechanical characteristics

Symbol	Parameter	Test conditions	Min.	Typ.(1)	Max.	Unit
FS	Measurement range(2)	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		

✎ Que signifient ces deux paramètres ?

✎ Quels sont ces `FS bits` cités dans les conditions ? Quelle est leur valeur dans notre cas ?

✎ Que doit-on en déduire et comment obtenir la valeur réelle d'accélération en g ?

💻 `@TODO (3)` : Construisez le nombre `z_value` dans l'unité demandé (en g).

🗗 Compilez, téléversez et observez le résultat sur terminal série. Confirmez que la valeur d'accélération évolue entre +1.0 g et -1.0 g selon l'orientation de la carte.

IV.3.f. Ajustement : pleine-échelle

Avec tout ce qui a été fait précédemment, vous avez pu déployer un firmware remplissant des fonctionnalités rudimentaires mais fonctionnelles. Ce composant offre de nombreuses fonctionnalités (notamment des machines d'état qui pourraient être utilisées pour des applications de casque VR, de souris gamer, …). Il n'est pas dans les objectifs de cette séquence pédagogique de faire la démonstration du plein potentiel de ce capteur. En revanche, nous pouvons encore peaufiner un détail.

💻 Épurez le code du `while(1)` du `main()` :

- modifiez la période de la boucle à 100 ms (au lieu de 1 s) ;
- commentez ou supprimez toutes les autres lignes, sauf la lecture et l'affichage de l'accélération sur l'axe z.

🗗 Validez sur terminal série que vous obtenez bien la valeur d'accélération.

🗗 Secouez la cible, vous devriez voir sur le terminal série que la valeur d'accélération sature.

✎ Quel registre et quels bits permettent de régler la sensibilité du capteur ? Quelles valeurs ces bits doivent-ils prendre pour obtenir la plus grande plage de mesure ?

💻 Terminez la définition de la fonction `ACCEL_2_init()` en imposant une nouvelle valeur de pleine-échelle au capteur (commentaire `@TODO (4)`).

🗗 Compilez, secouez, validez.

Mazette, la valeur d'accélération ne correspond plus ! En effet dans la fonction de lecture de l'accélération, le coefficient de sensibilité dépend de la plein-échelle de mesure. Au lieu de faire une fonction valable pour une seule sensibilité (et donc pour une seule pleine-échelle), il faut faire évoluer son code de sorte à rendre celle-ci adaptable.

💻 Dans la fonction `ACCEL_2_readZ()`, supprimez le code correspondant au `@TODO (3)`.

💻 Au `@TODO (4)`, effectuez une lecture de la valeur de pleine-échelle.

💻 Au `@TODO (5)`, testez la valeur de pleine-échelle afin de savoir quel coefficient doit être appliqué à la valeur d'accélération lue depuis les registres.

💻 Compilez, téléversez, testez, validez.

💻 Si ce programme de test est fonctionnel, complétez et validez les fonctions de lecture d'accélération des autres axes : `ACCEL_2_readX()`, `ACCEL_2_readY()` et `ACCEL_2_readXYZ()`.

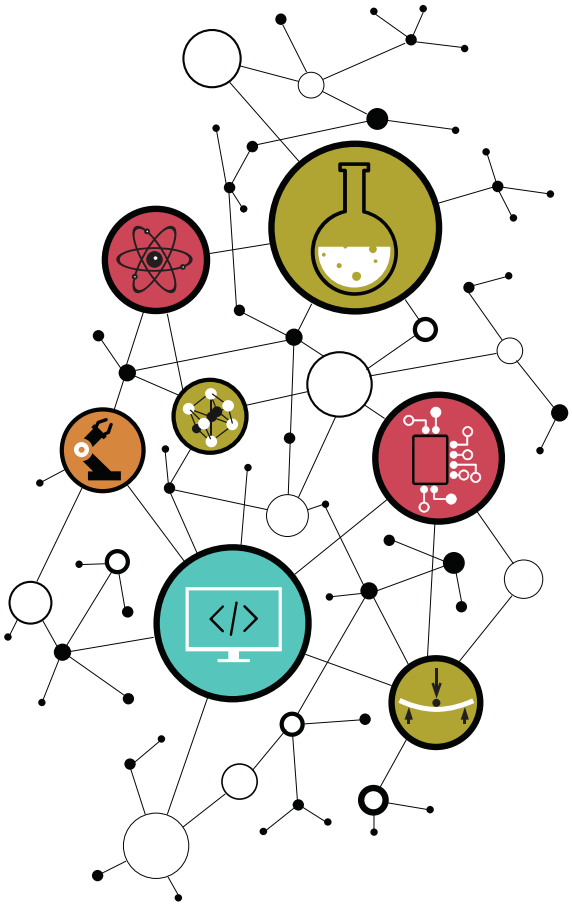
V. Synthèse

✎ En reprenant les différents points de cette partie, effectuez ici une synthèse de ce qui a été vu. Cette synthèse pourra être consultée lors des futurs développements.

1. Concernant la couche physique (fils, signaux, tensions, niveau logiques, …)
2. Concernant la couche protocole (nombre de bits, qui gère quelle ligne, …)
3. Concernant la couche firmware (configuration du périphérique, fonctions utilisées)
4. Concernant le côté capteur (mêmes paramètres ?, accès aux registres, …)

PART 4

BUS I²C



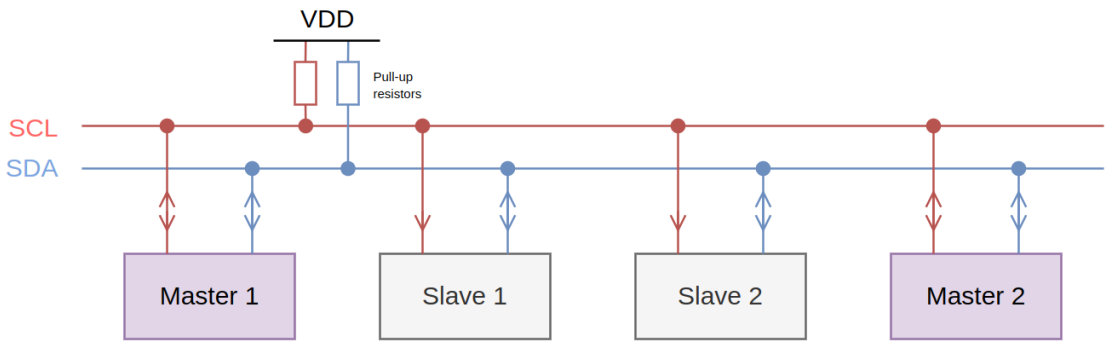
I. Description

Développé en 1982 par Philips Semiconductors (aujourd'hui NXP), le bus I²C est un bus de communication **série**, **synchrone**, **half-duplex** et basé sur une topologie **maître-esclaves**. Contrairement au SPI, l'I²C offre la possibilité d'avoir plusieurs maîtres sur le bus.

Le bus I²C est constitué de deux lignes bidirectionnelles :

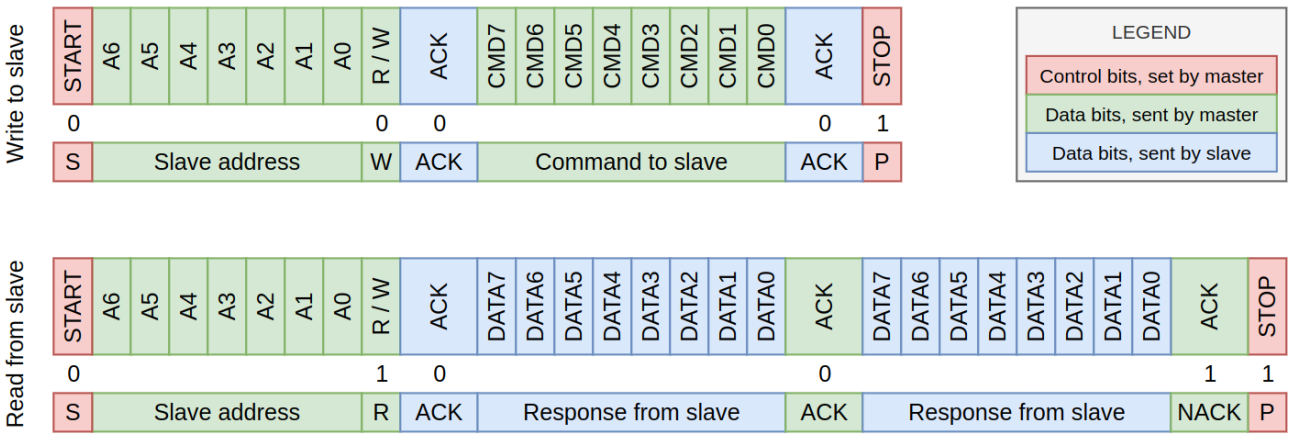
- `SCL` : Serial Clock, ligne manipulée uniquement par les nœuds maîtres ;
- `SDA` : Serial Data, manipulée tantôt par un nœud maître, tantôt par un nœud esclave.

Ces deux lignes sont tirées vers un niveau logique haut de par deux résistances de *pull-up*.



Le principal avantage de l'I²C face au SPI est son nombre réduit de fils : seulement deux sont nécessaires (alors que le SPI nécessite 3+*n* fils, *n* étant le nombre d'esclaves). En effet, la sélection de l'esclave en I²C ne se fait pas par le biais d'une ligne électrique, mais par un adressage des cibles.

Pour communiquer avec un esclave, le maître envoie d'abord un bit de start ('0'), 7 bits d'adresse désignant l'esclave et le mode d'accès (Read/Write), puis laisse l'esclave acquitter ('0'). Ensuite, en fonction du mode d'accès (Read/Write), la ligne SDA est manipulée soit par le maître soit par l'esclave adressé. Dans tous les cas, c'est toujours le maître qui décide d'arrêter l'échange avec un bit de stop (aussi, en cas de lecture, le maître n'acquitte pas la dernière trame).

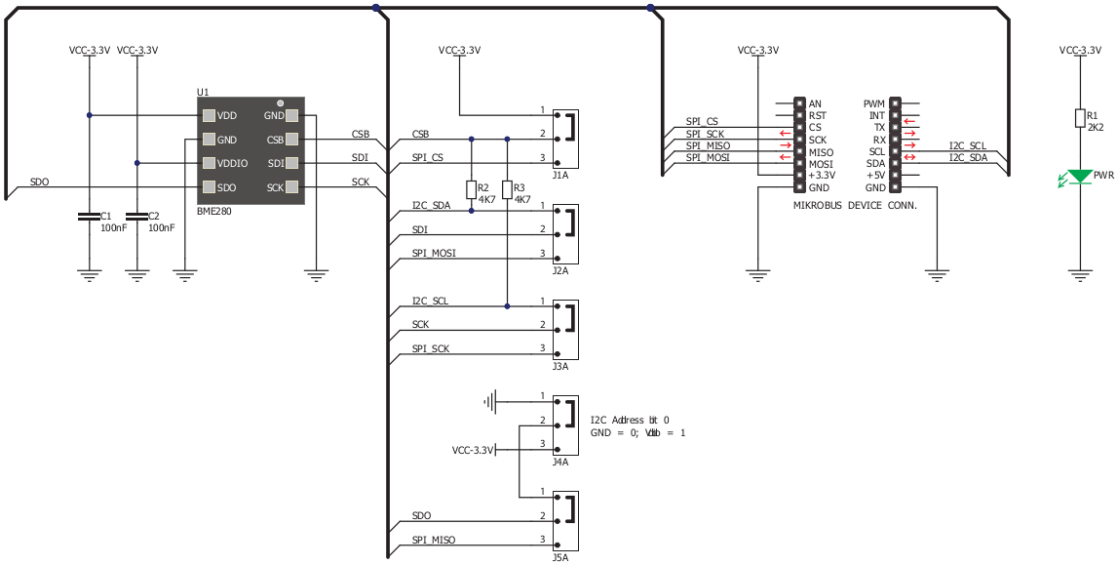


II. Application : Weather click

Nous allons étudier le protocole I²C au travers d'un nouveau Click Board : le Weather click.

II.1. Étude du capteur

Ici encore, le click board n'est qu'une simple interface de connectique pour le capteur embarqué (un Bosch Sensortec BME280), comme en témoigne son schéma très simple.



✎ Quelles sont les grandeurs mesurées par le capteur BME280 (voir sa datasheet) ?

✎ Sur cette carte, quels sont les signaux nécessaires à une communication I²C ?

✎ À l'aide du schéma de l'Arduino UNO click shield et du schéma de la carte NUCLEO, retrouvez sur quelles broches du MCU sont reliées les broches SPI du Light click.

✎ Trouvez le jumper J4A sur le schéma, et notez la position de ce jumper sur votre carte.

✎ Quel est le rôle des résistances R2 et R3 ? La section « 6.2 I²C interface » de la datasheet du BME280 (capteur du Weather click) peut aider.

II.2. Configuration du périphérique I²C

Comme vous avez pu l'observer, l'interface de communication utilisée avec le Weather click est un bus I²C. Configurons ce périphérique sur le micro-contrôleur STM32.

💻 Dans STM32Cube IDE, créez un nouveau projet STM32 (reprenez l'annexe si besoin).

💻 Vous aurez besoin des informations suivantes lors de la **création** du projet :

- `Board selector` : celle que vous avez en TP (voir au dos, référence NUCLEO-xxxxxx)
- `Project Name` : VOTRENOM_i2c ;
- `Project Location` : sélectionnez le répertoire `connectivity/workspace/i2c/` ;
- `Initialize all peripherals with their default Mode ?` : Yes.

💻 Vous aurez besoin des informations suivantes lors de la **configuration** du projet :

- Sur la vue composant, sélectionnez le mode I²C sur les deux broches repérées plus tôt
  - Cela vous permet d'identifier le périphérique I²C utilisé (I2C1, I2C2 ou I2C3)
- Connectivity → I2Cx
  - I2C : I2C
  - tout le reste par défaut

💻 Générez le code () et vérifiez que la fonction `main()` contient désormais l'appel à la fonction `MX_I2Cx_Init()`.

II.3. Driver weather_click

Comme pour les Click boards déjà rencontrés dans ce module, il vous faudra développer un driver permettant à un utilisateur d'échanger facilement des informations avec le Weather click. Les fichiers drivers se trouvent dans le répertoire `connectivity/workspace/drivers/`.

💻 Copiez et collez les fichiers `weather_click.c` et `weather_click.h` dans les répertoires du projet `…/CM4/Core/Src/` et `…/CM4/Core/Inc/` respectivement. Depuis l'explorateur de projet de l'IDE, vous allez sûrement devoir rafraîchir ces deux dossiers (F5) pour voir apparaître les fichiers.

En parcourant ces deux fichiers, vous devriez être maintenant assez familiers avec leurs contenus. Seulement cette fois-ci, on voit dans le fichier `weather_click.c` des fonctions déclarées et définies avec le mot-clé `static`.

🔪 Que signifie le qualificateur `static` pour une fonction et en C ?

II.3.a. Initialisation du handle

Lisez la documentation (commentaires Doxygen) du fichier `weather_click.h`.

✎ À quoi sert le champ `hi2c` de la structure `WEATHER_handle_t` ? Quelles broches (ou quels signaux) permet-il de manipuler ?

✎ Quel paramètre doit-on fournir à la fonction `WEATHER_init()` ?

✎ Retrouvez dans le `main.c` la déclaration du *handle* du périphérique I²C. Quel est son nom ?

✎ Vous n'avez fait que trouver sa déclaration, mais à quel endroit ce handle est-il initialisé ?

💻 Débutez la définition de la fonction `WEATHER_init()`, (commentaire `@TODO (1)`) en initialisant le *handle* avec le paramètre fourni à la fonction.

💻 Dans la fonction `main()`, trouvez l'endroit le plus approprié pour appeler la fonction d'initialisation du Weather click. Compilez et éliminez toute source d'erreur ou de warning.

Il n'y a pas lieu d'exécuter le programme sur cible puisque les fonctions de lecture et d'écriture n'ont pas encore été rédigées.

II.3.b. Fonction de lecture depuis le périphérique

✎ À partir de la datasheet du BME280, retrouvez les informations pour effectuer une lecture des registres du capteur depuis un bus I²C. Indiquez ici les étapes.

✎ Quelle est l'adresse de ***votre*** esclave ?

🖳 Définissez la macro-constante `WEATHER_I2C_ADDR` en conséquence.

✎ Dans le fichier driver du périphérique I²C fourni par la HAL, retrouvez et indiquez ici les deux fonctions de transmission et réception du périphérique I²C.

🖳 Avec ces deux fonctions de la HAL et les indications de la datasheet du BME280, complétez la définition de la fonction `WEATHER_readReg()` du fichier `weather_click.c`.

Pour tester la fonction de lecture de registre, nous allons lire la valeur du registre `id`.

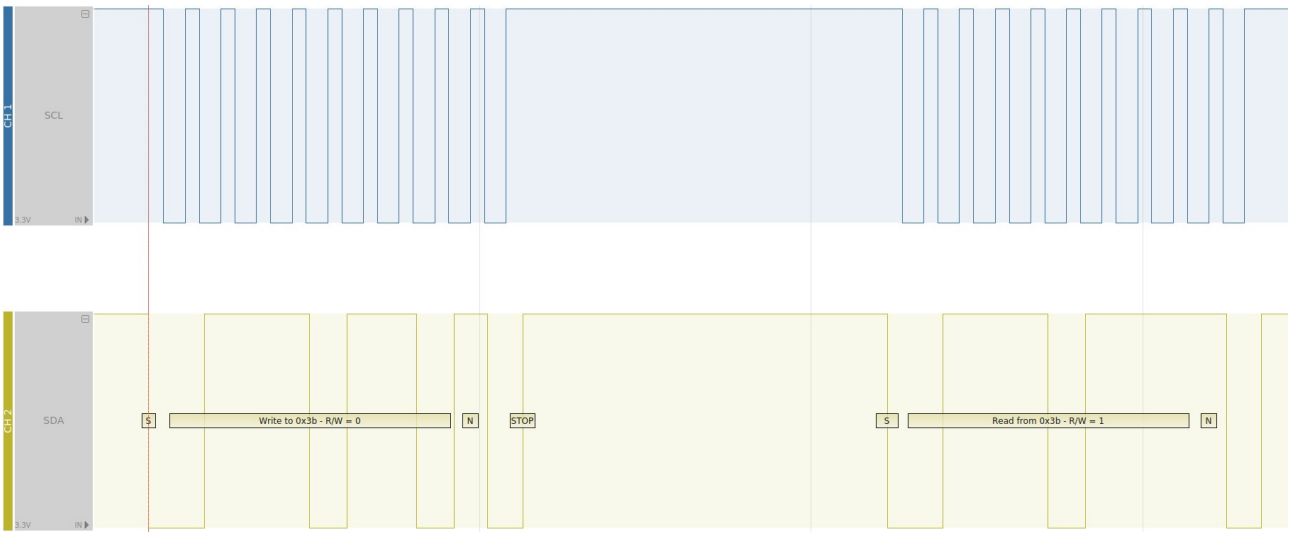
✎ Quelle est l'adresse de ce registre `id` ? À quoi sert-il ? Quelle devrait être la valeur lue ?

🖳 Testez la fonction `WEATHER_readReg()` dans le `main()` en lisant la valeur du registre `id`.

🗗 Affichez-la sur un terminal série.

La valeur lue ne semble pas être correcte (ou si elle l'est, c'est un heureux hasard).

Effectuez une capture avec un analyseur logique pour vérifier les trames échangées, et effectuez un décodage du protocole I²C sur ces trames. Voici ce que vous devriez observer.



Combien d'octets aurait-on dû observer d'après la figure 10 de la datasheet ? Détaillez.

À quoi correspondent les deux octets observés ?

Quelle est l'adresse décodée ici ? Est-ce celle vous avez demandé (celle de votre esclave) ? (*Note : les bits d'adresse (et de données) sont lus sur front montant de SCL).*

Que remarque-t-on si on compare ces bits d'adresse à ceux de la figure 10 de la datasheet ?

À quoi correspond le bit « N » ou « NACK » indiqué par le décodeur ? Est-ce normal ?

✎ Revenons aux fonctions de la HAL. Si vous lisez le cartouche Doxygen de la fonction de transmission I²C `HAL_I2C_Master_Transmit()`, qu'est-il indiqué pour le paramètre `DevAddress` ?

✎ Même question pour la fonction `HAL_I2C_Master_Receive()`.

En effet l'adresse donnée par le capteur est sur 7 bits (« *The 7-bit device address is 111011x* »). Mais pour correspondre au protocole I²C, l'adresse de l'esclave doit occuper les 7 bits de poids fort afin de laisser le bit 0 (LSb) indiquer s'il s'agit d'une écriture (valeur '0') ou d'une lecture ('1').

💻 Modifiez la définition de la macro-constante `WEATHER_I2C_ADDR` pour décaler les bits d'un rang vers les poids forts.

🗗 Effectuez de nouveau un test en affichant la valeur lue sur le terminal série.

🗗 Validez que la valeur affichée sur le terminal est bien celle attendue avant de poursuivre.

🗗 Effectuez une relevé à l'analyseur logique.

🗗 À l'aide du décodeur de trame et de la figure 10 de la datasheet, indiquez le rôle de chaque bit présent sur la ligne SDA, on précisant qui en est à l'origine (maître ou esclave). Soyez aussi complet que possible.

Ce que vous devez constater est que la ligne SDA peut être manipulée par le maître et l'esclave, mais que chacun a un slot temporel fixé (il ne peuvent pas manipuler la ligne en même temps). Cela signifie que le protocole I²C est **half-duplex**.

Plusieurs composant peuvent utiliser la même ligne SDA car celle-ci est par défaut tirée vers l'état logique haut grâce à une résistance de pull-up, et toutes les sorties SDA des circuits intégrés sont en open-drain.

✎ Comment un composant écrit-il un '0' logique ? Un '1' logique ?

✎ Quel est le niveau logique d'un acquittement ? Pourquoi ce n'est pas le contraire ?

II.3.c. Fonction d'écriture vers le périphériques

✎ À partir de la datasheet du BME280, retrouvez les informations pour effectuer une écriture dans les registres du capteur depuis un bus I²C. Indiquez ici les étapes.

🖳 Avec ces deux fonctions de la HAL et les indications de la datasheet du BME280, complétez la définition de la fonction `WEATHER_writeReg()` du fichier `weather_click.c`.

Nous testerons et validerons la fonction d'écriture par la suite. Nous allons d'abord déterminer les paramètres à imposer au BME280 pour répondre à notre cahier des charges, puis nous écrirons ces valeurs dans les registres concernés.

II.3.d. Configuration du BME280

Comme de nombreux capteurs, et comme le LIS3DSL (accéléromètre) étudié dans la partie précédente, le BME280 est pourvu de registres permettant de configurer le capteur à la guise de l'utilisateur.

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

✎ Repérez dans la datasheet les trois registres de contrôle à configurer.

✎ Dans un de ces registres, le champ `mode[1:0]` doit être paramétré. Indiquez les trois modes de fonctionnement possibles. Lequel permet de réaliser un relevé périodique des grandeurs physiques ?

✎ Pour les trois registres de contrôle, précisez les valeurs des bits en suivant le cahier des charges suivant :

- relevé des trois grandeurs physique, sans sur-échantillonnage
- relevé périodique, toutes les 500 ms
- pas de filtre

Maintenant que les paramètres de contrôle ont été déterminés, passons à la configuration du composant en complétant la définition de la fonction `WEATHER_init()`.

💻 La première chose à faire quand on veut paramétrer ce type de composant est de vérifier que celui-ci est bien joignable. Procédez à l'identification du capteur en suivant le commentaire `@TODO (2)` (vous pourrez par la suite supprimer le code équivalent dans la fonction `main()`).

💻 Procéder à la configuration du BME280 en écrivant les valeurs déterminées plus haut dans les registres correspondants. Prenez garde à l'ordre d'écriture dans les registres (cf. datasheet).

💻 Enfin, effectuez la lecture des paramètres de calibration du capteur[13] en appelant la fonction `WEATHER_getCalibParams()` au commentaire `@TODO (4)`. Cette fonction vous est fournie avec sa définition complète.

13 Les paramètres de calibration seront expliqués dans la section « Part 4 - II.3.e » de ce document.

La fonction d'initialisation maintenant écrite, nous allons la tester et la valider. Cela permettra en même temps de confirmer le fonctionnement de la fonction `WEATHER_writeReg()`.

💻 Dans le `main()`, appelez la fonction `WEATHER_init()` en début d'application.

💻 Dans la boucle principale de l'application, écrivez le code permettant de lire la valeur des registres configurés.

🗗 Affichez leur valeur sur le terminal, puis comparez à la valeur attendue.

🗗 Une fois les fonctions d'initialisation et d'écriture du BME280 validées, effectuez une capture à l'analyseur logique. Vous devez observez l'écriture d'une valeur dans un registre.

🗗 À l'aide du décodeur de trame et de la figure 9 de la datasheet, indiquez le rôle de chaque bit présent sur la ligne SDA, on précisant qui en est à l'origine (maître ou esclave). Soyez aussi complet que possible.

Ne passez à la suite que si ce point est validé.

II.3.e. Lecture des grandeurs mesurées

Afin de maîtriser les différents paramètres pouvant impacter la lecture de la température, de la pression et de l'humidité, une lecture de la datasheet est nécessaire. En effet ce composant est en réalité plus complexe qu'il n'y paraît. Ainsi la lecture des parties citées ci-dessous est conseillée pour répondre aux questions qui suivent :

- 3.4 Measurement flow (jusqu'au 3.4.3 inclus)
- 4. Data readout
- 5.4 Register description (pour les registres que vous lirez)

✎ Quel est l'intérêt du sur-échantillonnage ?

✎ Quel est l'intérêt d'utiliser un filtre (ici à réponse impulsionnelle infinie) ?

✎ Quelle est la résolution des valeurs d'humidité, de pression et de température dans notre cas (pas de filtre, pas de sur-échantillonnage) ?

✎ Quels registres contiennent le résultat des mesures (température, pression et humidité) ?

✎ Le nombre de bits sur lesquels sont stockées les valeurs correspond-il à leur résolution ? Que devrait-on normalement faire si ces nombres ne correspondent pas ?

✎ Après avoir récupéré le contenu des trois registres qui forment la valeur de température, comment reconstruire la valeur de température sur une seule variable de type entier ?

La valeur ainsi construite ne fournit cependant pas la valeur vraie de température. Il est en effet stipulé que chaque élément de mesure est différent (sous-entendu, d'un composant à l'autre). En conséquence, les paramètres de calibration de chaque BME280 ont été déterminés en usine puis stockés en dur dans la mémoire NVM (*Non-Volatile Memory*) du composant. Ces paramètres, accessibles depuis les registres, permettent de « compenser » les valeurs mesurées pour calculer les valeurs vraies.

La datasheet donne le code des fonctions de compensation des trois valeurs (température, pression, humidité) dans la section « 4.2.3 Compensation formulas ». Dans ces extraits de code, les valeurs sont stockées sur des entiers 32-bit en utilisant une représentation en virgule fixe (« *fixed-point arithmetic* »). Bosch Sensortec fourni aussi ces fonctions sur son repository GitHub[14] avec des valeurs au format flottant (« *floating-point* »).

Dans le fichier `weather_click.c` les fonctions de compensation ont été ré-écrites, afin de correspondre au reste du driver (prototypes de fonctions, valeurs, structures, ...). Parmi les deux versions proposées, c'est celle effectuant les opérations en virgule fixe qui a été choisie.

✎ Pourquoi devons-nous effectuer les calculs sur des entiers et non sur des flottants ? *Indice : ceci est lié à notre MCU*[15].

✎ Qu'est-ce qu'un nombre en virgule fixe ? Quels sont les avantages ? Recherchez sur Internet ou auprès de l'enseignant.

14 https://github.com/boschsensortec/BME280_driver/tree/master
15 Cette question n'est pas forcément juste selon le STM32 utilisé …

Observez les cartouches Doxygen des trois fonctions de compensation présentes dans le fichier `weather_click.c`. Parmi les informations données, on y trouve le format et l'unité des valeurs retournées `fine_temp`, `fine_press` et `fine_hum`)[16].

✎ Proposez un pseudo-code de conversion de la valeur de température `fine_temp` (format entier 32-bit, exprimé en 0.01 °C) en `temp_degC` au format flottant exprimé en °C.

✎ Proposez un pseudo-code de conversion de la valeur d'humidité `fine_hum` (format virgule fixe Q22.10, en % RH) en `hum_rh` au format flottant en %RH (pourcentage d'humidité relative).

✎ Proposez un pseudo-code de conversion de la valeur de pression `fine_press` (format virgule fixe Q24.8, en Pa) en `press_hPa` au format flottant en hPa.

Avec toutes ces informations, il est maintenant possible de compléter la définition des fonctions de lecture des grandeurs mesurées.

💻 Complétez la définition des fonctions `WEATHER_getTemperature()`, `WEATHER_getPressure()` et `WEATHER_getHumidity()` en suivant le raisonnement des questions-réponses ci-dessus.

💻 Complétez aussi la définition de la fonction `WEATHER_getAll()`, effectuant simplement un appel aux trois fonctions précédentes.

💻 Testez ces fonctions dans le `main()`, grâce aux lignes de code suivantes :

```
    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 );
```

🗗 Validez avec le terminal série.

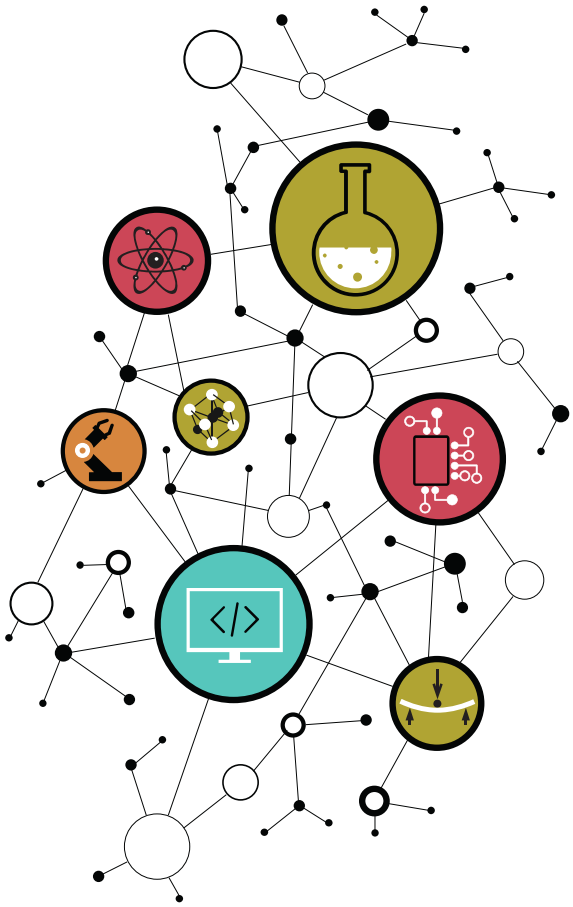
16 Ces informations sont évidemment issues de la datasheet, section « 4.2.3 Compensation formulas ».

II.4. Synthèse

✎ En reprenant les différents points de cette partie, effectuez ici une synthèse de ce qui a été vu. Cette synthèse pourra être consultée lors des futurs développements.

1. Concernant la couche physique (fils, signaux, tensions, niveau logiques, …)
2. Concernant la couche protocole (nombre de bits, qui gère quelle ligne, …)
3. Concernant la couche firmware (configuration du périphérique, fonctions utilisées)
4. Concernant le côté capteur (mêmes paramètres ?, accès aux registres, …)

PART 5
GLOSSARY



LSb / MSb	*Least/Most Significant bit*
LSB / MSB	*Least/Most Significant Byte*
MCU	***Microcontroller Unit***
CPU	*Central Processing Unit*
Peripheral	Internal hardware component of the MCU, designed for a specific task
GPIO	*General Purpose Input/Output*, peripheral that controls the MCU pins
STM	***STMicroelectronics***, 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	*Hardware Abstraction Layer*, functions for an easy use of MCU peripherals
STLink VCP	*STLink Virtual Com Port,* accessible through the USB link of the NUCLEO
NVIC	*Nested Vector Interrupt Controller*, ARM CPUs interrupt controller
IF	*Interrupt Flag*, indicates that an event has occured
IRQ	I*nterrupt Request*, asks the CPU to pause its execution
ISR	*Interrupt Service Routine*, function executed when an IRQ stops the CPU
Callback	Function called by an ISR and which definition must be written by the dev
UART	***Universal Asynchronous Receiver-Transmitter***, serial comm peripheral
USART	*Universal Synchronous-Asynchronous Receiver-Transmitter*
Tx	*Transmit line*
Rx	*Receive line*
SPI	***Serial Peripheral Interface***
	Bidirectional, full-duplex, synchronous serial bus, designed by Motorola
MISO	~~*Master*~~ *Main In,* ~~*Slave*~~ *Sub Out*
MOSI	~~*Master*~~ *Main Out,* ~~*Slave*~~ *Sub In*
$\overline{SS}$	*Slave Select*, active low
$\overline{CS}$	*Chip Select*, active low
I²C	***Inter-Integrated Circuit***
	Bidirectional, half-duplex, synchronous serial bus, designed by Philips
SDA	*Serial Data Line*
SCL	*Serial Clock Line*
ACK	*Acknowledgment*
NACK	*Non-Acknowledgment*