UMONS
Université de Mons

Faculty of Engineering

Project

Hardware and Software

Théo ABRASSART
Louis HUBERT

POLYTECH MONS

SEMI Electronics & Microelectronics

Under the leadership of Professor Carlos VALDERRAMA
and Mister Daniel BINON

February-June 2019

pôle hainuyer

Contents

List of Figures

Acknowledgements

Many thanks to D. Binon, A. Quenon and C. Valderrama for their technical assistance and guidance throughout the project to make sound decisions.

Chapter 1

Theoretical recall

1.1 Project presentation

As part of our Master 1 project for the hardware and software class, we have to use a microcontroller to control an ultrasonic sensor via the I2C bus. It is the SRF02 sensor that can measure the distance between this one and an object. The main objective is to match the programming part with the circuit.

1.1.1 Distance detection

The first application is the detection of the object distance. The principle is similar to the echo-rental used by bats or dolphins to locate themselves in space. The principle is simple, we send a mechanical wave (ultrasound) into the air. We know that the speed of sound in the environment is calculated as follows:

$$v = \sqrt{\gamma RT} \tag{1.1}$$

With γ the Laplace coefficient, R the ideal gas constant and T the air temperature. Air at our temperature and pressure condition is considered an ideal gas.

Therefore, by knowing the speed of the sound in the air, with a simple variation of time (round trip) we can know the distance of an object. Either d the distance from an object:

$$d = v\frac{\Delta t}{2} \tag{1.2}$$

1.1.2 Calculation of volume, area or perimeter

By correctly performing the measurements we could imagine calculating the volume, area or perimeter of a room. All you have to do is measure the xyz dimensions of the part and then apply the mathematical formulas.

1.2 Product presentation

During this project, we will use the STM32F303RE Nucleo 64. This chip is part of a series of microcontrollers produced by ST. The 32 represents the number of bits of the integrated circuit. The term Nucleo refers to the compatibility with Arduino, this series has been designed for this purpose. To be able to interact with this chip, we will install 3 programs including the following descriptions and installation steps. Don't forget to install the driver to communicate with the computer and the chip.

1.3 Ultrasound sensor

The ultrasound sensor is the SRF02. The picture is represented on the figure 1.1.

Figure 1.1: SRF02 sensor [1]

Important information about this sensor are mentioned in the datasheet. For example, the address of each register can be found in the figure 1.2 :

Location	Read	Write
0	Software Revision	Command Register
1	Unused (reads 0x80)	N/A
2	Range High Byte	N/A
3	Range Low Byte	N/A
4	Autotune Minimum - High Byte	N/A
5	Autotune Minimum - Low Byte	N/A

Figure 1.2: Datasheet - Address of registers [2].

As you can see, the Command Register have 0x00 as address. The CR is the area where the user have to write the name of the commands. There are a lot of commands which can be written on the CR. The figure 1.3, which comes from the datasheet, present all of these.

The command 0x50 to 0x52 is used to extract some information about the measurement (the distance in inches, centimetres or the round trip time in microseconds).

There are some commands named ”fake...” (0x56 to 0x58), these ones are used to test the communication with the sensor by checking the constant value (”fake”) which is stocked in the sensor's memory.

To extract these measurements, as mentioned in the picture 1.2, there are two 8 bits register, 0x02 and 0x03. The final measurement is on 16 bits. The tips is to concatenate these registers with the register 0x02 which has the most significant bits (MSB) and the register 0x03 that has the least significant bits (LSB).

Remark : the commands on the picture 1.3 are only available for the I2C communication. There are other commands in the datasheet which are only for the serial port but there are not presented on this tutorial. The serial port was deliberately destroyed by the former users of our sensor.

1.3.1 Device address

To start, it is necessary to know the precise address of the sensor in order to communicate with it. The address of the sensor is noticed on this one. But be careful, this address is a

Command		Action
Decimal	Hex	
80	0x50	Real Ranging Mode - Result in inches
81	0x51	Real Ranging Mode - Result in centimeters
82	0x52	Real Ranging Mode - Result in micro-seconds
83	0x53	Real Ranging Mode - Result in inches, automatically Tx range back to controller as soon as ranging is complete.
84	0x54	Real Ranging Mode - Result in centimeters, automatically Tx range back to controller as soon as ranging is complete.
85	0x55	Real Ranging Mode - Result in micro-seconds, automatically Tx range back to controller as soon as ranging is complete.
86	0x56	Fake Ranging Mode - Result in inches
87	0x57	Fake Ranging Mode - Result in centimeters
88	0x58	Fake Ranging Mode - Result in micro-seconds
89	0x59	Fake Ranging Mode - Result in inches, automatically Tx range back to controller as soon as ranging is complete.
90	0x5A	Fake Ranging Mode - Result in centimeters, automatically Tx range back to controller as soon as ranging is complete.
91	0x5B	Fake Ranging Mode - Result in micro-seconds, automatically Tx range back to controller as soon as ranging is complete.
92	0x5C	Transmit an 8 cycle 40khz burst - no ranging takes place
93	0x5D	Get software version - sends a single byte back to the controller
94	0x5E	Get Range, returns two bytes (high byte first) from the most recent ranging.
95	0x5F	Get Minimum, returns two bytes (high byte first) of the closest range measurable - see Autotune section
96	0x60	Force Autotune Restart - same as power-up. You can ignore this command.
160	0xA0	1st in sequence to change I2C address
165	0xA5	3rd in sequence to change I2C address
170	0xAA	2nd in sequence to change I2C address

Figure 1.3: Datasheet - List of commands [2]

PIC address on 8 bits. The I2C requires a 7 bits address and the eighth bit is a 1 or a 0 just to know if the process is writing or reading.

So, the address was 0xE0, which means in binary : 1110.000, if we move every bits to the right we get 0111.0000 which equal to 0x70 in hexadecimal. The 0x70 address is the good one for the STM32 with I2C communication.

1.4 I2C

The communication interface between the sensor and the microprocessor is I2C. It is not necessary to create the driver allowing discussion between those elements since an I2C library is already present on the project created by CubeMX. However, before using the I2C, it's important to know how it works. We will therefore introduce the principle of operation as well as some important nuances to understand well.

1.4.1 Important information

The I2C communication needs a start and stop condition. There are 3 signals : the clock (SCL), the data (SDA) and finally the signal reference (bulk) which means 3 wires.

- Start condition : SDA pass to 0 while SCL is equal to 1;
- Stop condition : SDA pass to 1 while SCL stays to 1.

Notice : the rest state is at 1. Indeed, the I2C protocol uses some resistors to allow to pull-up the lines. When the device needs to transmit a 0, then he short-circuits the line through a pull-down transistor (like a *NPN* or a *Mosfet*).

After checking if the bus is free and takes control of it, the chip becomes the master. The signal clock is imposed by him. The I2C sequence is on 9 bits :

- 7 bits for the device address;
- 1 bit for reading or writing;
- 1 bit of an ACK.

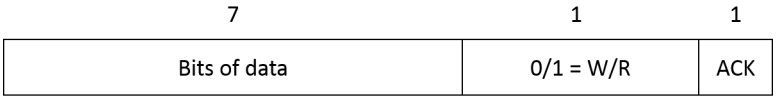


Figure 1.4: I2C sequence.

1.4.2 Byte transmission

We present here the steps of the transmission of a sequence of bits :

- The master transmits the MSB bit on the D7 on SDA ;
- He agrees this one by putting a 1 on SCL ;
- When SCL passes to 0, he does the same with D6 while D0 ;
- After that, he sends an ACK at 1 while observing the value of SDA ;
- To report the good running, the slave imposes a 0 ;
- The master sees this 0 and can continue with others sequences.

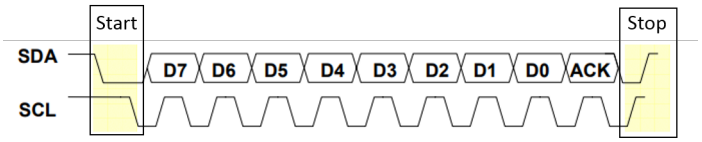


Figure 1.5: I2C transmission [6].

1.4.3 Data reading

We present here the steps of the reading of a sequence of bits :

- The master sends the address of the device (7 bits) and waiting for the ACK;
- The ACK is placed by the slave and then he puts the data on the SDA wire;
- The master can continue or stop the reading by setting a 0 or a 1 respectively.

1.5 Interrupt

Interrupt is a fundamental mechanism for micro-controllers. It allows events outside the micro-controller to be taken into account and to associate them with specific processing called routine. It should be known that an instruction in progress is never interrupted, because in reality it is at the end of the execution of this one and at the arrival of the event that the interrupt management routine is executed [6]. Here is the classical steps of the course of an interruption :

- detection of the triggering event (defined thanks to the functions referred in appendix A) ;
- end of the current instruction ;
- identification of the event occurred ;
- saving the address of the current instruction in a stack ;
- traversal to the interrupt processing routine (created by the user in a specific file, at a specific place and associated with interrupt) ;
- treatment of the corresponding interrupt (in the routine) ;
- retrieving the return address (stack removal) ;
- Return to the initial program.

Remark : a rule of good practice is to perform short operations with an interrupt. This is inadvisable to do heavy operations like a big for loop or others.

Chapter 2

Tools Installation

2.1 Presentation of the tools

We will use the CubeMX interface that will allow us to configure a project by downloading the latest libraries available on the web. From CubeMX, we simply have to select the chip which is the STM32 Nucleo 64 F303RE. So when we generate the code, it is done on the Eclipse AC6 platform previously installed. We therefore have a C workspace with its own functions. To be able to track from a console, you must also install the software *Terra Term Pro*.

2.2 CubeMX

??

CubeMX is the software that will allow us to generate the project from the microprocessor previously chosen in it. The used chip is the STM32F303RE Nucleo 64.

2.2.1 Installation

To install CubeMX you have to :

- obtain the executable of this software, go to the website *https://www.st.com/en/development-tools/stm32cubemx.html* $<$ *Get Software*;
- provide your name/first name/e-mail address to be able to have 24 hours of download access on the site;
- run the .exe ;
- accept the general conditions of use.

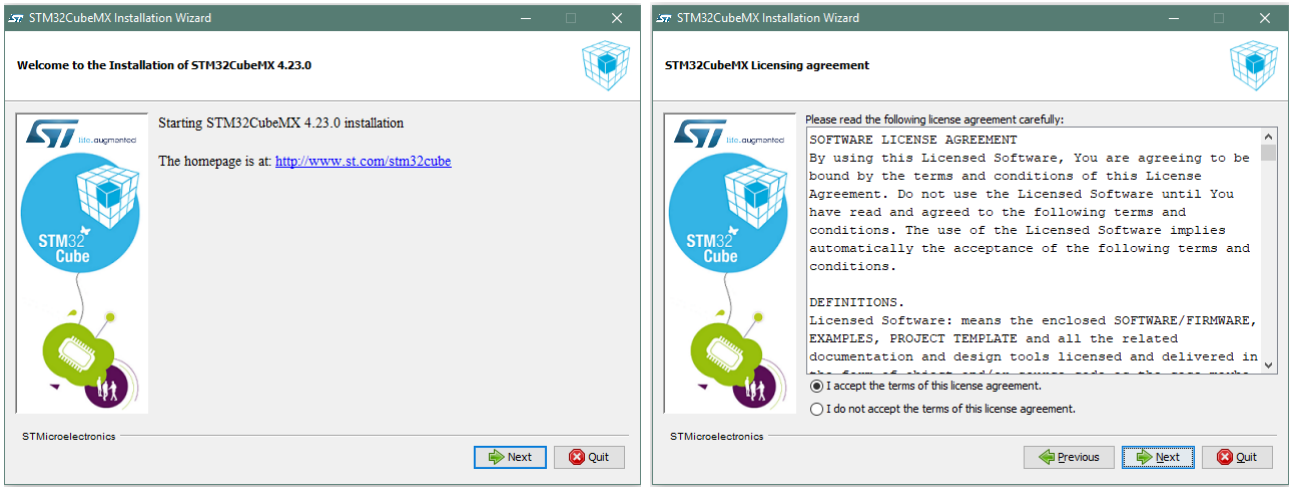


Figure 2.1: First steps in CubeMX's installation [3].

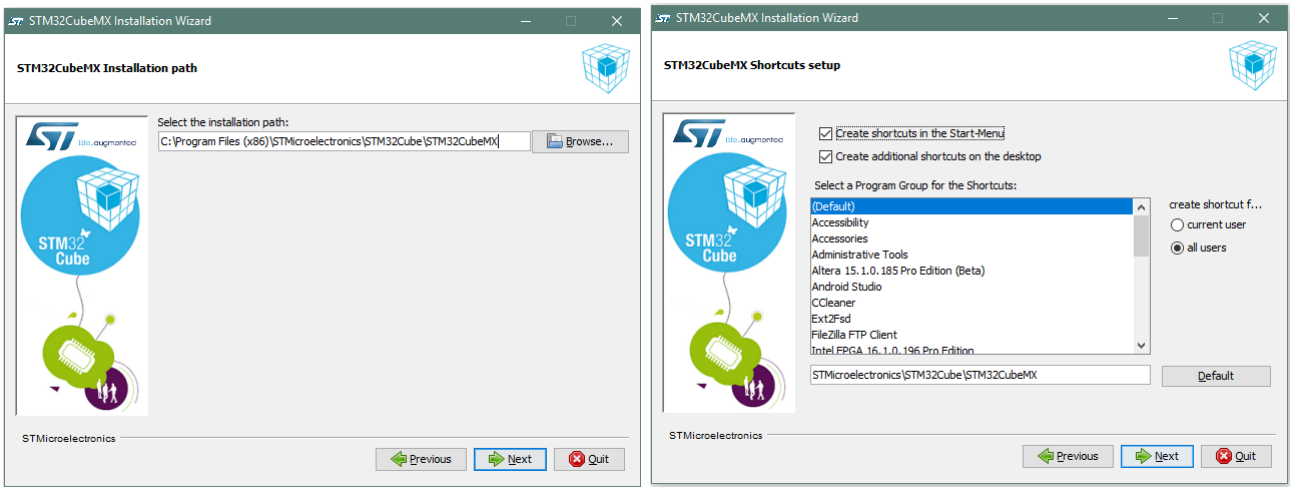


Figure 2.2: Second steps in CubeMX's installation [3].

2.3 AC6

[4]

AC6 is an Eclipse software that will allow us to implement the code we will establish (in C) to the STM32 chip after compilation.

2.3.1 Installation

To install this software, you have to :

- get the software to : https://www.st.com/en/development-tools/sw4stm32.html < Go to
- launch the executable file;
- press the *Next* button at each step
- accept the general conditions of use when necessary.
- determine where the directory will be stored.

- at the end (in the last figure), a summary of all checked options is available before installation.

  After that a window will open indicating the progress of the installation. Once finished, you will have access to the program and you can start your C code on this software.

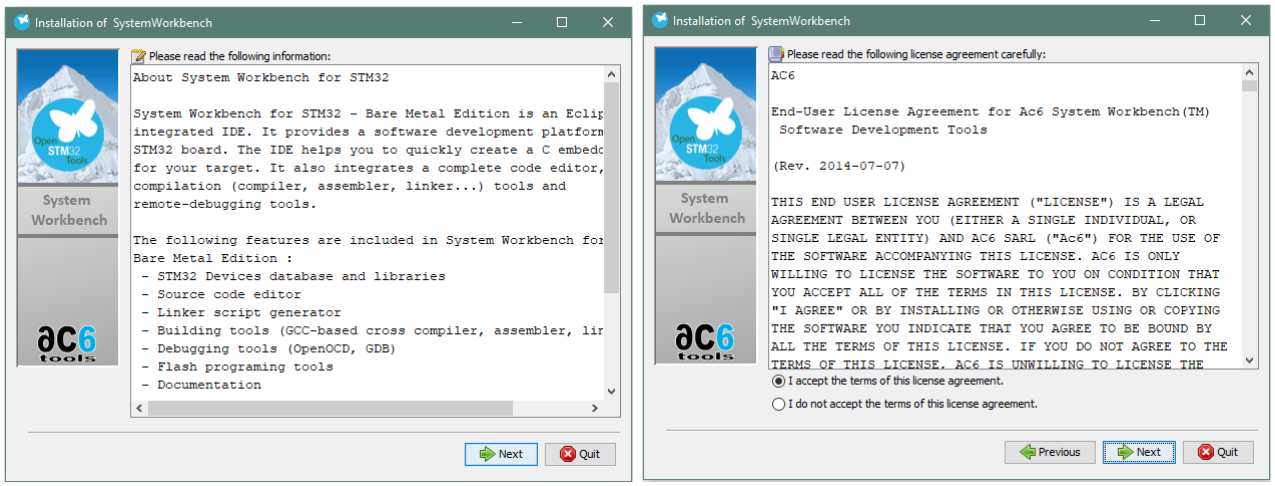


Figure 2.3: First steps in *AC6*'s installation [4].

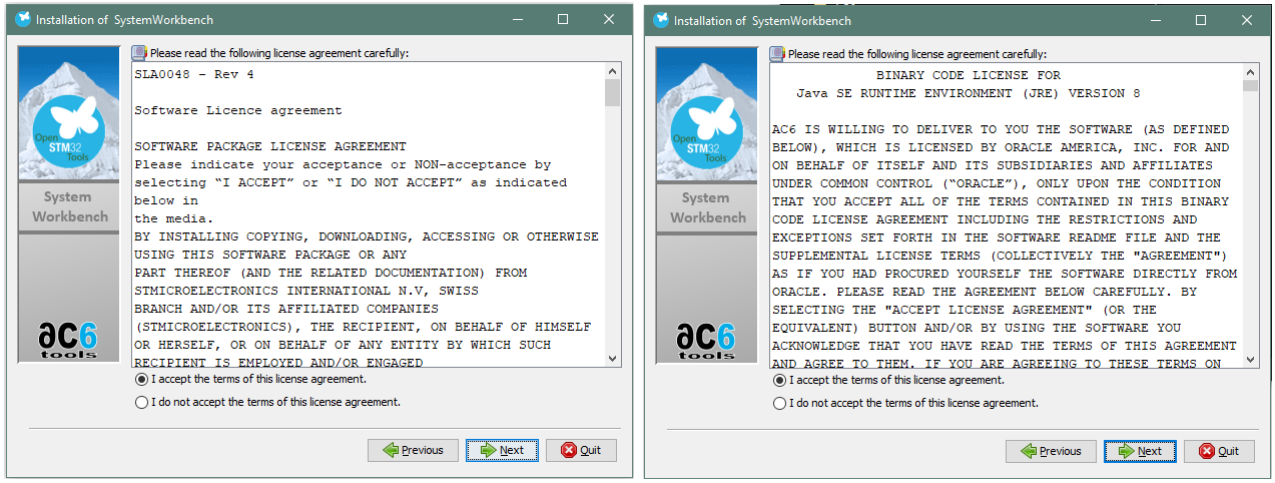


Figure 2.4: Second steps in *AC6*'s installation [4].

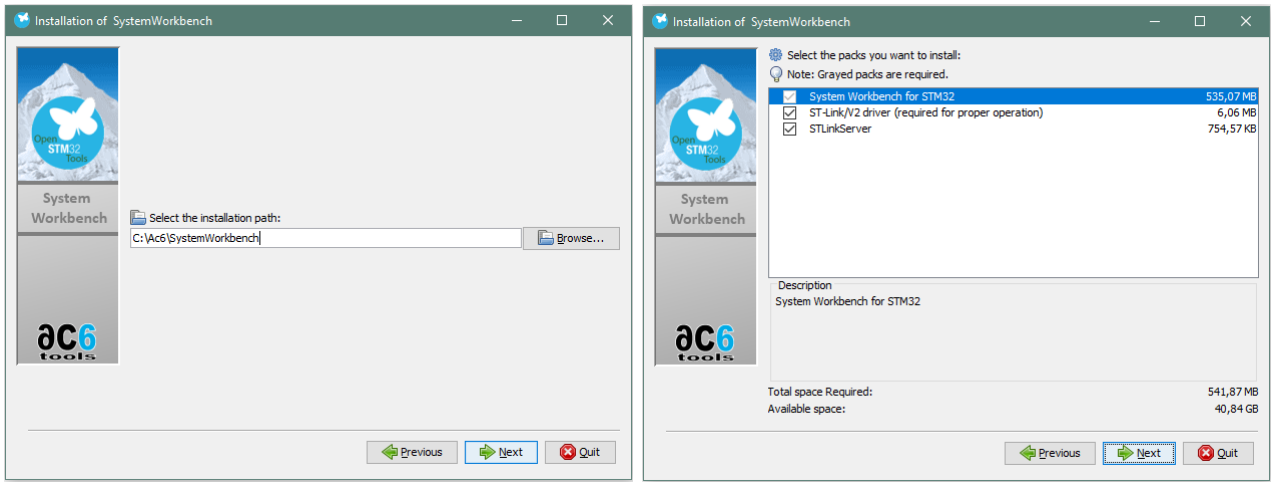


Figure 2.5: Third steps in *AC6*'s installation [4].

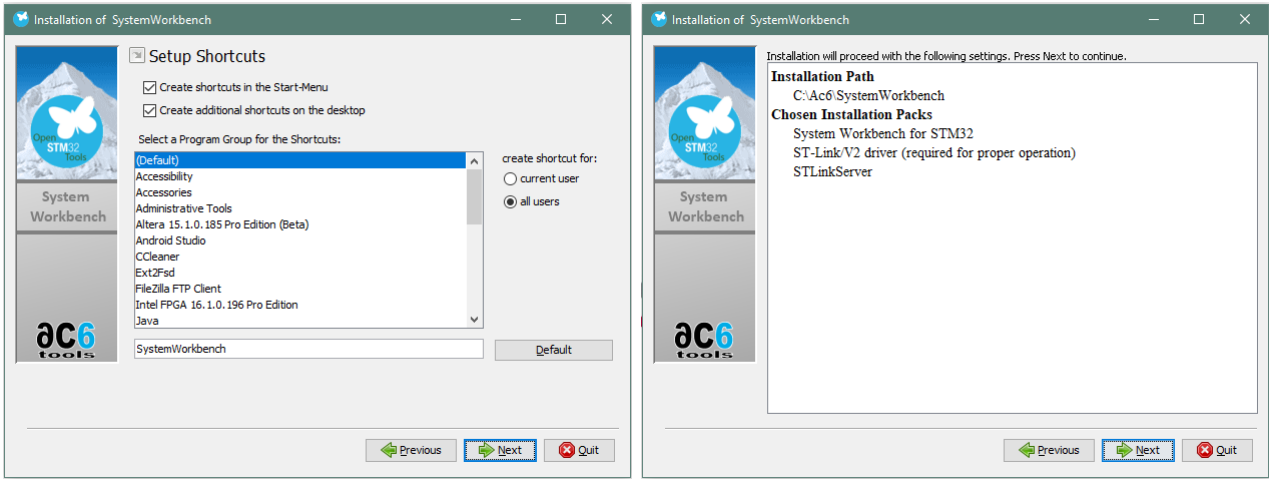


Figure 2.6: Fourth steps in *AC6*'s installation [4].

2.4 Terra Term Pro

[5]

Terra Term Pro is a software that will allow the display of a console to interact with the chip when it is connected through the USB port on a computer. If you do not have an LCD display, this console will be necessary to locate you in the process.

2.4.1 Installation

To install this *Tera Term Pro* you have to :

- download the executable file from the site: *https://ttssh2.osdn.jp/index.html.en* < *Download* < *Download package* < *Most recent version*;
- choose the language correctly;
- accept the terms and conditions, the Terra Term Pro software will be installed.

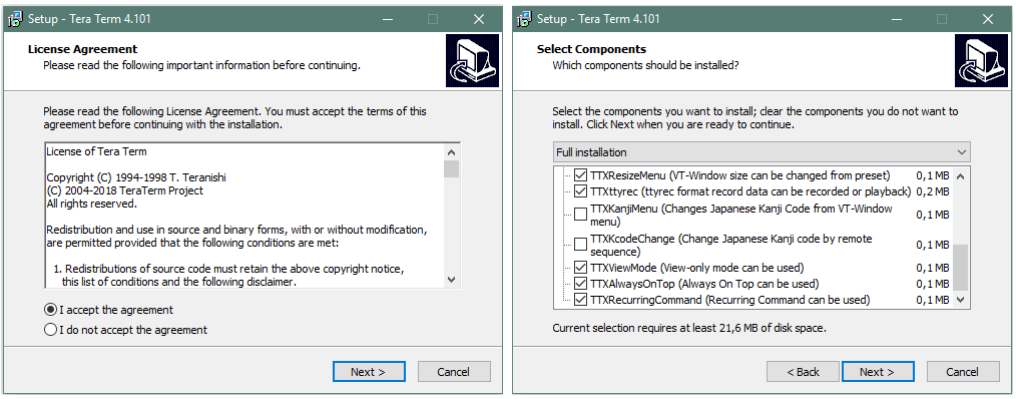


Figure 2.7: First steps in *Tera Term Pro*'s Installation [5].

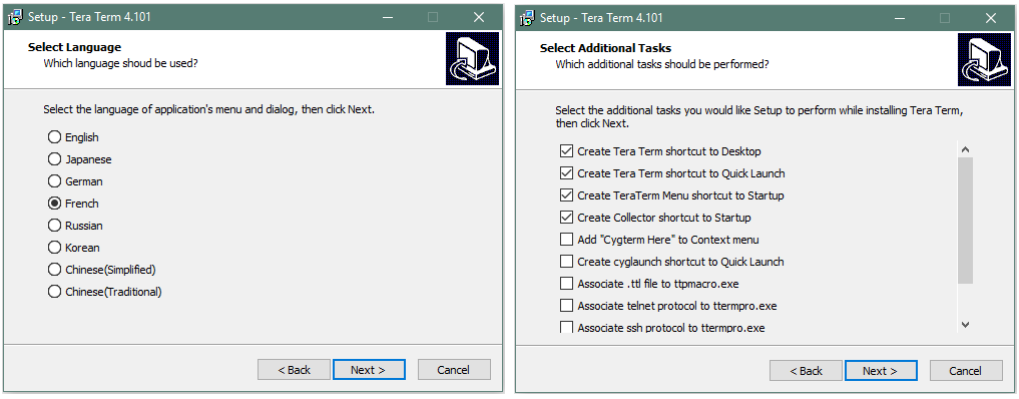


Figure 2.8: Second steps in *Tera Term Pro*'s Installation [5].

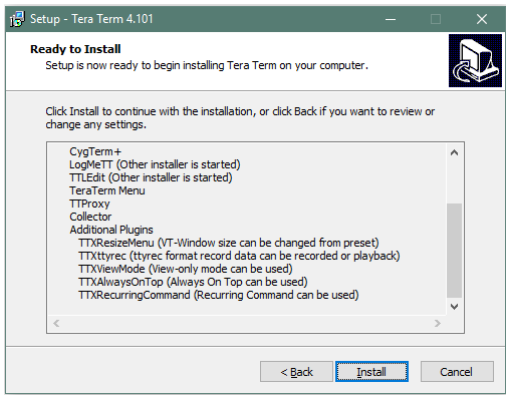


Figure 2.9: Third steps in *Tera Term Pro*'s Installation [5].

Chapter 3

Software development

3.1 Tools configuration

This section will introduce how to create a project by configuring the necessary drivers using the *CubeMX* software. We will then see how to implement a code using the *AC6* software and how to compile it and send it on the map. Finally, we will see how to display the results using the serial port using the *Tera Term Pro* software.

3.1.1 Drivers configuration

The configuration of the drivers is done through the *CubeMX* software. To do it, you have to :

- Open *CubeMX* whose interface is placed on the fig. 3.1. You can see on the left of the interface, the two main sections to open an already existing project or to create a new one. In our case, we will create a new project, to do this we must choose the button *access to board selector*, framed in red dotted, which allows us to access a representation of our chip ;

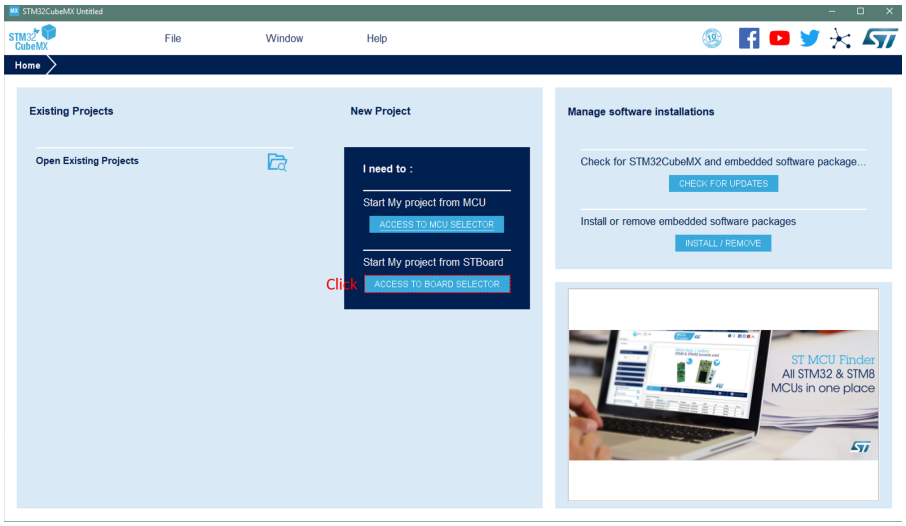


Figure 3.1: *CubeMX* interface [3].

- Next, select the type of chip we are using, in our case the Nucleo64 F303RE chip, using the selection filters framed in red dotted on the fig. 3.2 ;
- Then, you arrive on the graphical interface of the chip used in the section *Pinout & Configuration*. Various tabs are available as you can see on the fig. 3.3 ;

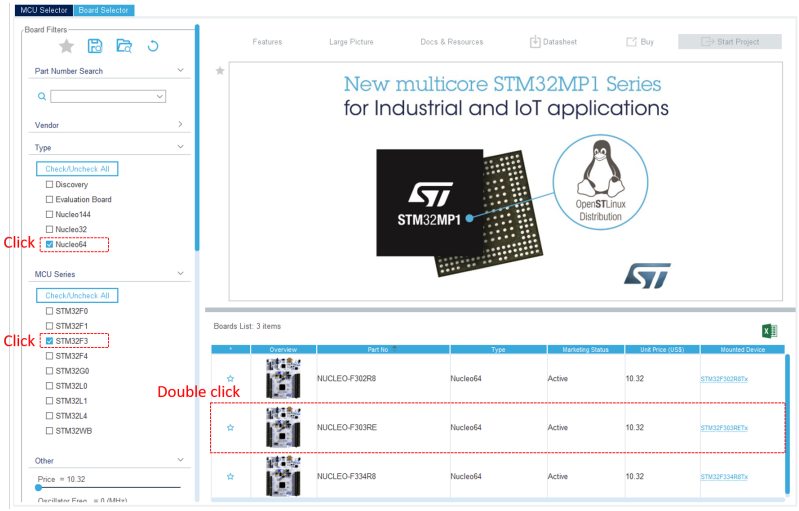


Figure 3.2: Selection of the wanted chip [3].

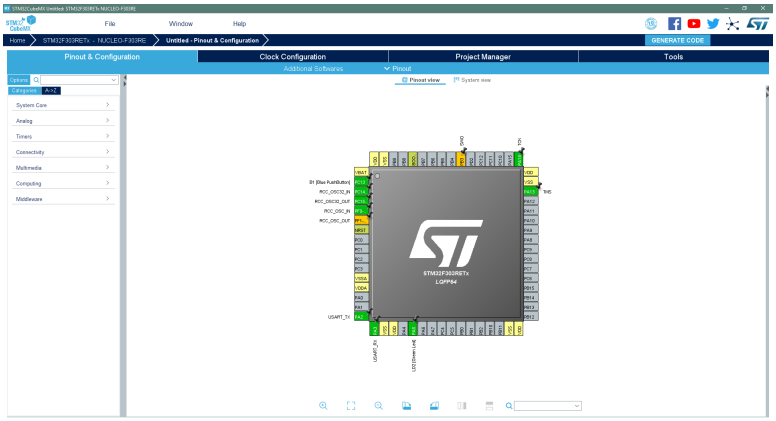


Figure 3.3: Graphical interface of the wanted chip [3].

- We notice that some pins on the have different colors. The green color indicates that the pin is used and that the function used is activated. For example, the *PA5* pin is used to control the LED on the map. The orange colored pins indicate that the pins are reserved for certain functions but that these functions are not activated. In order to configure a function of a pin, take for example the *PB8* and *PB9* pin. Let's click on it to display all the functions available via these pins. In order to select a click on the desired function, in this case we want to control an I2C, choose functions framed red dotted on the fig. 3.4 ;

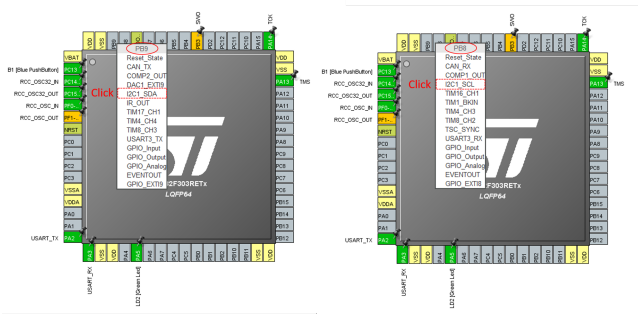


Figure 3.4: Choose the functions of the pins [3].

- Once the functions selected, the pins appear in orange (framed in red on the fig. 3.5). To activate the functions, go to *Connectivity* section, click on the type of I2C chosen, in our case the 1, which will be by default on *disable* and select I2C as shown on the fig. 3.6 ;

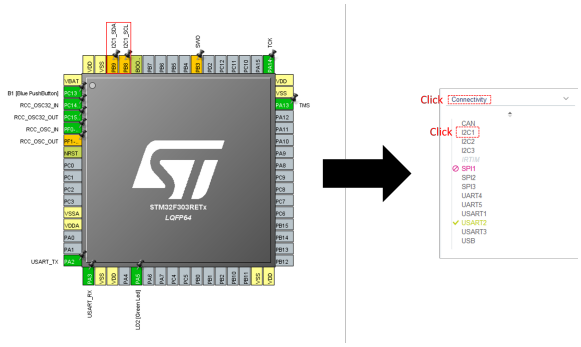


Figure 3.5: Orange pins to be activated [3].

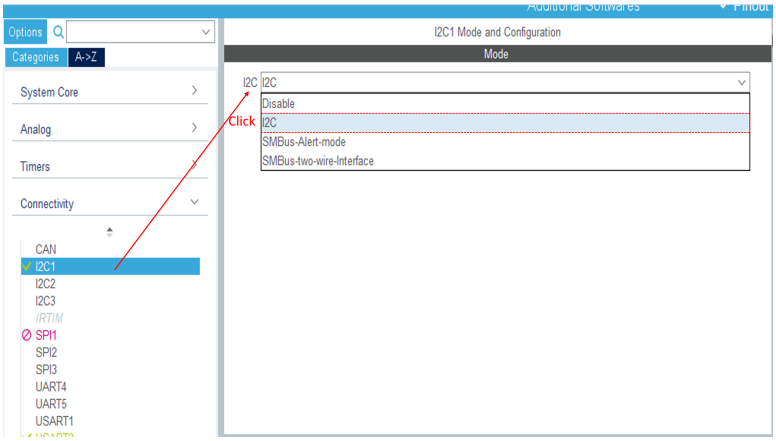


Figure 3.6: Orange pins to be activated [3].

- Then, go to the *Project Manager* section shown on the fig. 3.7, and specify the project name, the path where you want to create the project, and the compiler which is not missing the compiler *EWARM V8* but which must be changed to have the compiler *SW4STM32* as shown on the fig. 3.8 ;

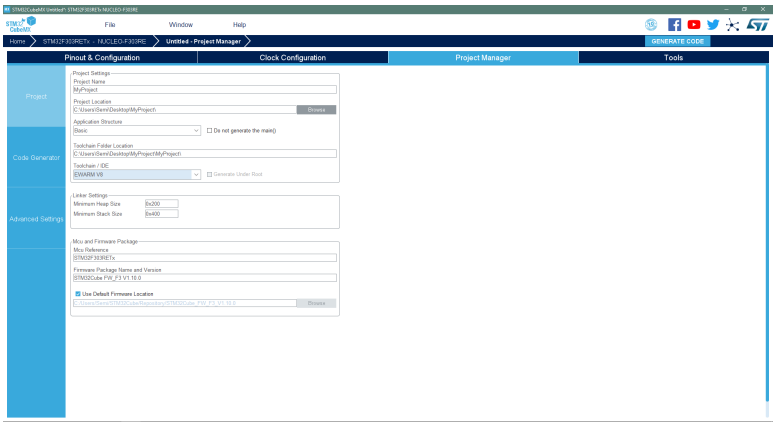


Figure 3.7: *Project Manager* interface [3].

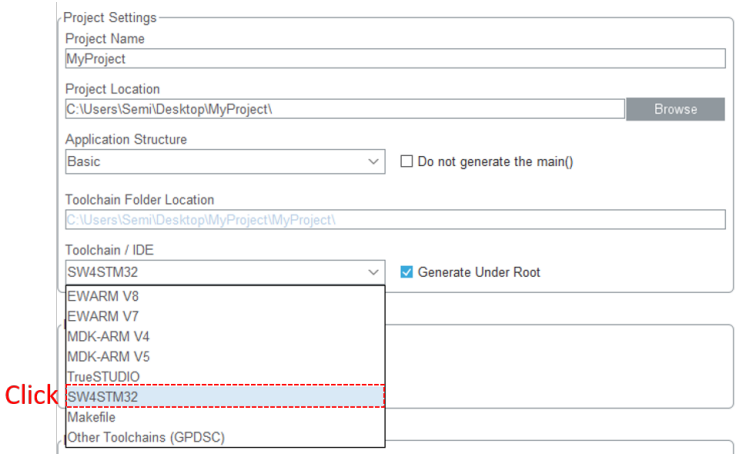


Figure 3.8: Choice of the compiler [3].

- Finally, press *Generate code* to open the desired compiler, in this case *AC6*. Once the creation of the project is done, the pop-up window shown on the fig. 3.11, appears, then click on *Open Project*.

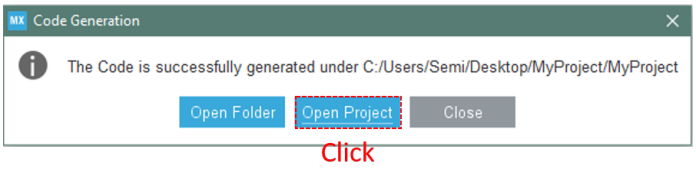


Figure 3.9: Open the project [3].

3.1.2 Code configuration

The *AC6* software opens. **It is important not to delete the lines of codes already present because they were put by *CubeMX* to allow to control the chip according to the configurations indicated during the previous step. You can edit and add your code where the comments tell you.** The body of your code is included in the infinite loop *while (1)* as shown on the fig. 3.10. On the left side of this window is the set of folders grouped under the project name. The *.c* files are in the *Src* folder.

Once your code has been set you can, helping you with the fig. 3.11 :

- Back up all your files (you should not see any asterisks next to the names of your open folders at the top of the space where your code is located) ;
- Then, you can put the little hammer that will compile your code and show you in the console display located on the bottom, if there are any errors ;
- If you want, you can use the debug function to see step by step how the code behaves ;
- Finally, once you are sure of your code, you can place the code on the map by clicking on the white arrow on green circle. Please note that the chip must be connected via USB to the computer, and if there is any problem with the upload, refers to the section 3.1.3.

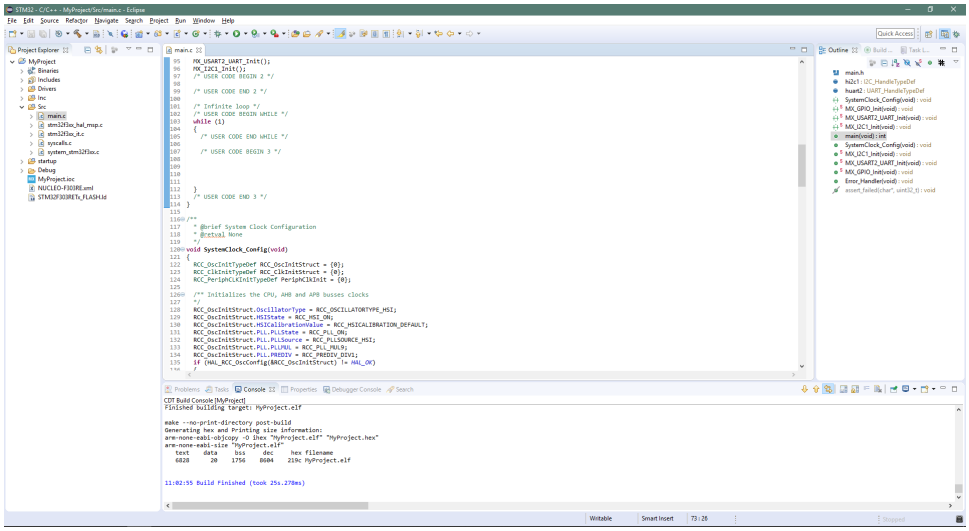


Figure 3.10: Interface of the *AC6* software, centred on the infinite loop.

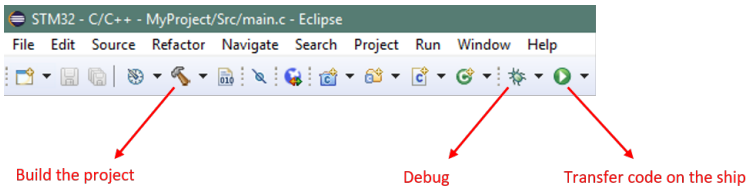


Figure 3.11: Commands of the *AC6* [4].

3.1.3 USB Communication

It is possible during the upload of the code in the map that windows indicates when the USB connection of the chip that it is not recognised. In order to solve the problem :

- Check if the chip is known or not, to do this go in *device Manager* < *USB bus controller*. If the chip is not recognised, then you will see a caution sign next to the device as shown on the fig. 3.12 ;

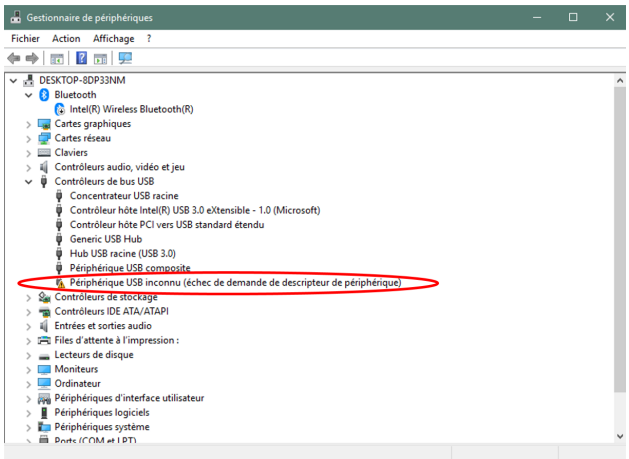


Figure 3.12: Chip not recognised.

- In this case, go to the producer's website, *STMicroelectronics*, and download the

drivers for your chip type[1].

- Launch the driver installation ;
- Go back in *device Manager* < *USB bus controller* and check that your device is recognised as shown on the fig. 3.13.

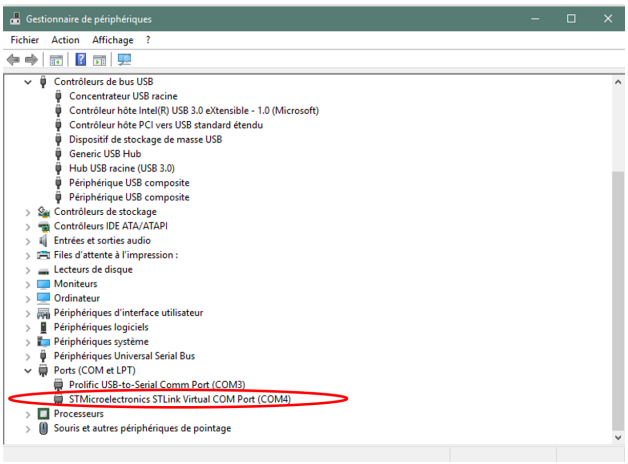


Figure 3.13: Chip not recognised.

3.1.4 Serial port configuration

Now that we know how to send information to the chip, it would be interesting to be able to read information that sends us the chip by USB connection. To do so, we will use the *Tera Term Pro* software that will allow us to read the data transiting on the USB port. The interface is on the fig. 3.14. So you have :

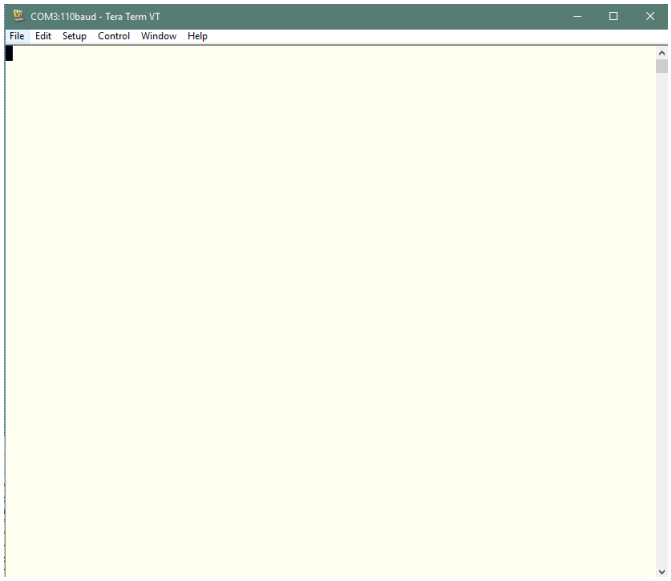


Figure 3.14: *Tera Term Pro* interface [5].

- to find on which USB port is your chip, to refer to you the *device Manager* < *USB bus controller* like in the section 3.1.3 on the fig. 3.13. In that case, we can see that

[1]In our case, the direct link to our chip type is .

the chip uses the port *COM 4*, so we have to tell to *Tera Term Pro* to use this port by going in *Setup* < *Serial port...* and set the *COM 4* like on the fig. 3.15;

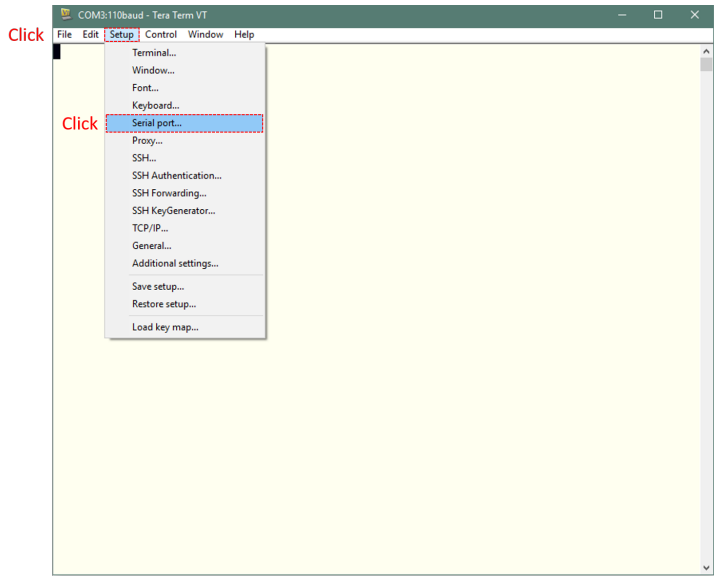


Figure 3.15: Configuration of the serial port [5].

- You also need to inquire the baud-rate. In our case, you find it on *CubeMX* on the section *Categories* < *Connectivity* < *UART2* < *Configuration*. So we can notice that the baud-rate is shown (cfr. fig. 3.16), and worths $38\,400\,\mathrm{Bits\,s^{-1}}$. It is obvious to go into the *UART* section because the UART protocol is used through the USB port;

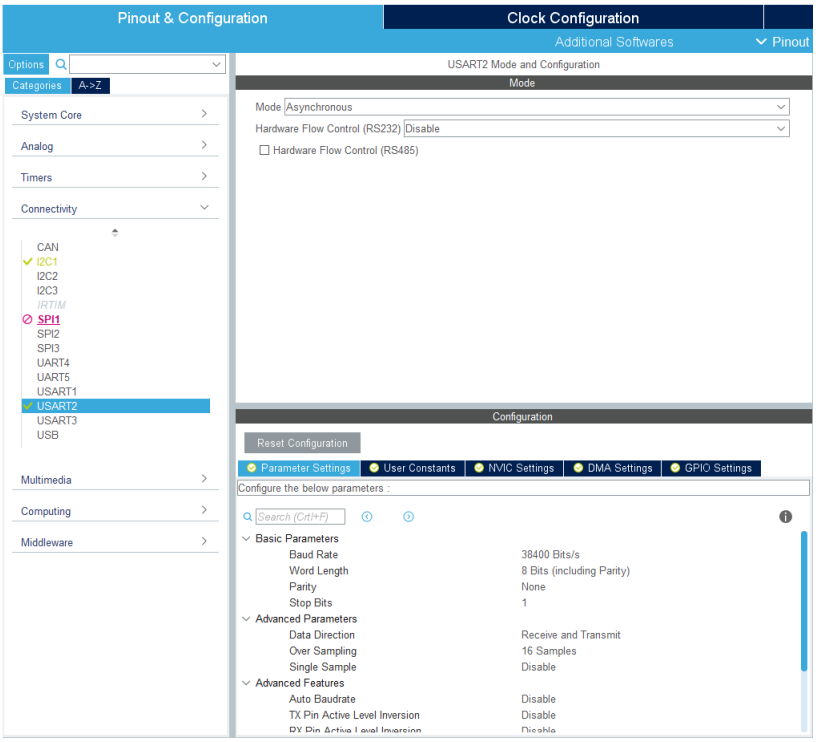


Figure 3.16: *CubeMX* baud-rate [3].

- configure the baud-rate of *Tera Term Pro* to $38\,400\,\mathrm{Bits\,s^{-1}}$ in the section *Categories* < *Connectivity* < *UART2* < *Configuration* like shown on the fig. 3.17;

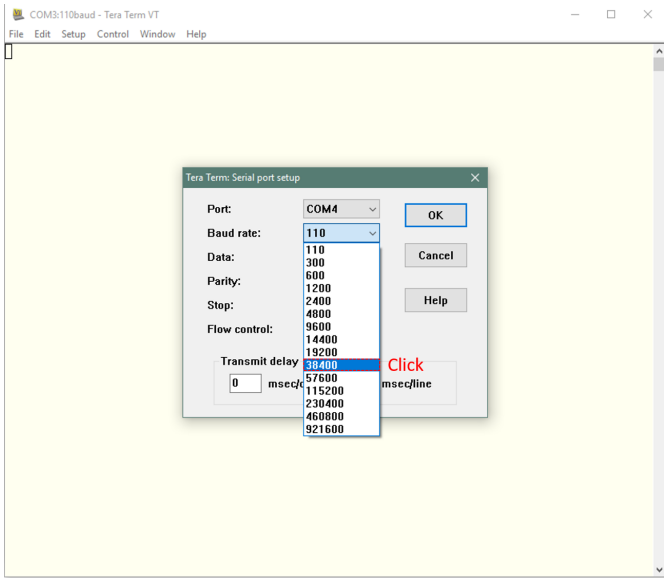


Figure 3.17: *Tera Term Pro* baud-rate [5].

- Finally, we must add these lines of code to the place indicated by the comments :

```
/* USER CODE BEGIN 0 */
#ifdef __GNUC__
 #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
 #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif

 PUTCHAR_PROTOTYPE{
  HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 100);
  return ch;

 }
/* USER CODE END 0 */
```

Indeed, the *printf* command sends all characters placed as an argument at once. The advantage of placing these lines of code is to send the set of characters placed in the *printf* argument, one by one in UART communication via the serial port.

3.2 Unit Testing

This section will present the different unit tests performed in order to test code directly on the target board. Indeed, the goal is to test some codes to better understand the operation of the chip, but also to practice to implement simple realisations before tackling a larger code. Thereby, we will discuss some simple implementations by presenting the code test and verifying that the chip reacts as expected. Note that for a better understanding of the codes, in addition to a brief explanation of them, the codes used will be commented.

3.2.1 LED control

As a first step, you must configure the chip to use the desired pins, to do this :

- you need to do setup the pin *PC13* and *PA5* (that are already green, so set-up) ;
- this the only thing to do because there are not particular function to implement because all the component are already on the chip.

Basically, there are 3 ways to drive a LED. The first method consists of direct writing of the state of the LED. The address of the latter is already known by the chip since the LED is on it. We will therefore go directly to the LED by imposing a state. The value 1 corresponds to the on state while the 0 corresponds to the off state. We can therefore alternately write 1 and 0 in the *while* infinite loop so as to make it blink by imposing a delay between each write. So here is the code used :

```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin,0);  // Write 0 on the led = turn off
   HAL_Delay(250);         // Delay of 250ms
   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin,1);  // Write 1 on the led - turn on
   HAL_Delay(250);
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

It is important to leave comments that are not on the lines where there is code, because they allow the compiler to generate the code. Without these comments, a multitude of errors may appear. We notice that the function used in this case is the **HAL_GPIO_WritePin** function which is used to write the state of the pine. For more information or clarification, the arguments of this function are explored in appendix A.

A second solution is to come to get the state of the LED and then write the opposite state. It's a more compact way than the previous one. To do this, we must remember that the oppose of a state is by means of the inverter *!*. Here is the code used :

```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, !HAL_GPIO_ReadPin(LD2_GPIO_Port, LD2_Pin)); // Write
       the inverse of the led's state
   HAL_Delay(500);                 // Delay of 500ms
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

To recover the state of the pin, the **HAL_GPIO_ReadPin** function is used. The arguments of this function are also explained in appendix A.

Finally, a third solution is to directly use the function associated with the change of state of the LED. This is the **HAL_GPIO_TogglePin** function. This code is the simplest and the most compact. Here is the code used :

```
 /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

   HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin); // Toggling the led
   HAL_Delay(50);         // Delai of 50 ms

    }

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
```

3.2.2 Button control

Like the button is directly on the chip, Now that the control codes of a LED have been tested, it is interesting to play with the only button available on the STM32. Thereby, using a **Switch Case**, we will control the LED following the 3 ways presented in the previous section. Each time the button is pressed, a *choix* variable is incremented. This one is in a *Modulo* operator so that when the value of the variable is greater than the number of desired cases, it returns to 0. We have in our situation 3 cases, the **Modulo** chosen so is the **Modulo 3**[2]. This implementation of incrementation and **Modulo** is performed in the first lines of the *while* loop.

```
1   while (1)
2   {
3     /* USER CODE END WHILE */
4   if(!HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)) {
5    choix = ((choix+1)%3);
6    HAL_Delay(10);
7   }
8
9   switch (choix){
10
11  case 0 :
12   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, 0);
13   HAL_Delay(250);
14   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1);
15   HAL_Delay(250);
16   break;
17
18  case 1 :
19   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, !HAL_GPIO_ReadPin(LD2_GPIO_Port, LD2_Pin));
20   HAL_Delay(500);
21   break;
22
23  case 2 :
24   HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
25   HAL_Delay(50);
26   break;
27  }
```

Thus, each time the user presses the button, the case changes and therefore the blinking rate of the LED also changes. We notice that in this case, we have to use a variable, *choix*, which must be declared at the beginning of the code as shown below :

```
1 /* USER CODE BEGIN PV */
2 int choix;
3 /* USER CODE END PV */
```

3.2.3 I2C unit testing

In order to ensure that we are able to communicate directly between the chip and a sensor using the I2C communication protocol developed in section 1.4, we will now test to send data using this protocol. However, the verification of the proper functioning of the communication protocol is more difficult than in the case of the LED, because in the case of the LED it was sufficient to check that the latter was flashing. In fact, based on the theoretical description of the I2C protocol, we will connect an oscilloscope to the two communications cables on which the data and the clock are travailing to verify that the signals on these cables are the ones we want to send. To do this :

- the assembly of the fig. 3.18 has been realised and the oscilloscope probes were plugged into the two cables of the chip. Note that the blue cable represent the data signal and the yellow cable represent the clock signal ;
- Check if the pull up is selected (depending of your choice, you can unselect it and make it manually). To do this, go to *Categories* < *I2C1* < *GPIO Settings* and activate the pull-up option like done on the fig. 3.19 ;

[2]As a reminder, in C, the modulo operation is represented by %.

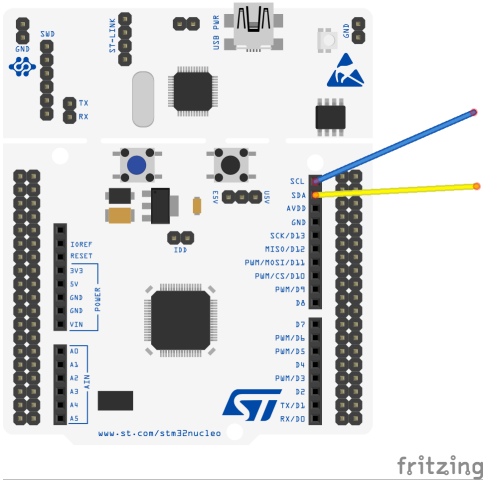


Figure 3.18: I2C test configuration (made with Fritzing).

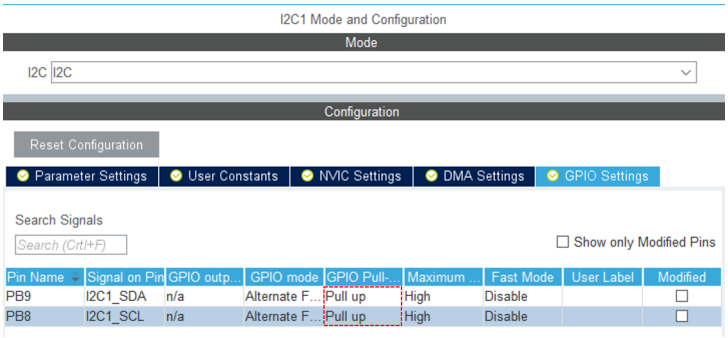


Figure 3.19: Activation of the pull-up [3].

- Finally, import the following code into the chip :

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    buffer[0]=0x00;            // Test to point the CR
    HAL_I2C_Master_Transmit(&hi2c1, 0x70<<1 , buffer, 1, 100); // Transmit the 0
    HAL_GPIO_TogglePin(GPIOA,LD2_Pin);       // Toggle de Pin
    HAL_Delay(250);            // Delay of 250 ms

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

In this code, we can notice the appearance of the function **HAL_I2C_Master_Transmit** that is used to carry out a transmission of the master, that is to say the chip to the slave, that is to say the sensor. So we notice that we use the first I2C of the chip (among the 3 that are available), and that we send a communication to address 0x70 whose bits are shifted by 1 on the left. So we have as sensor address . The information sent is the buffer array (declared at the top of the code where the declarations are intended), and from this buffer array only the first box is sent. Note also that a timeout of 100ms has been set in case of no response from the sensor. The blinking of the LED has also been put to see if the code has been correctly imported on the chip.

The waveform obtained when an oscilloscope is connected to the two cables used by the I2C are on the fig. 3.20.

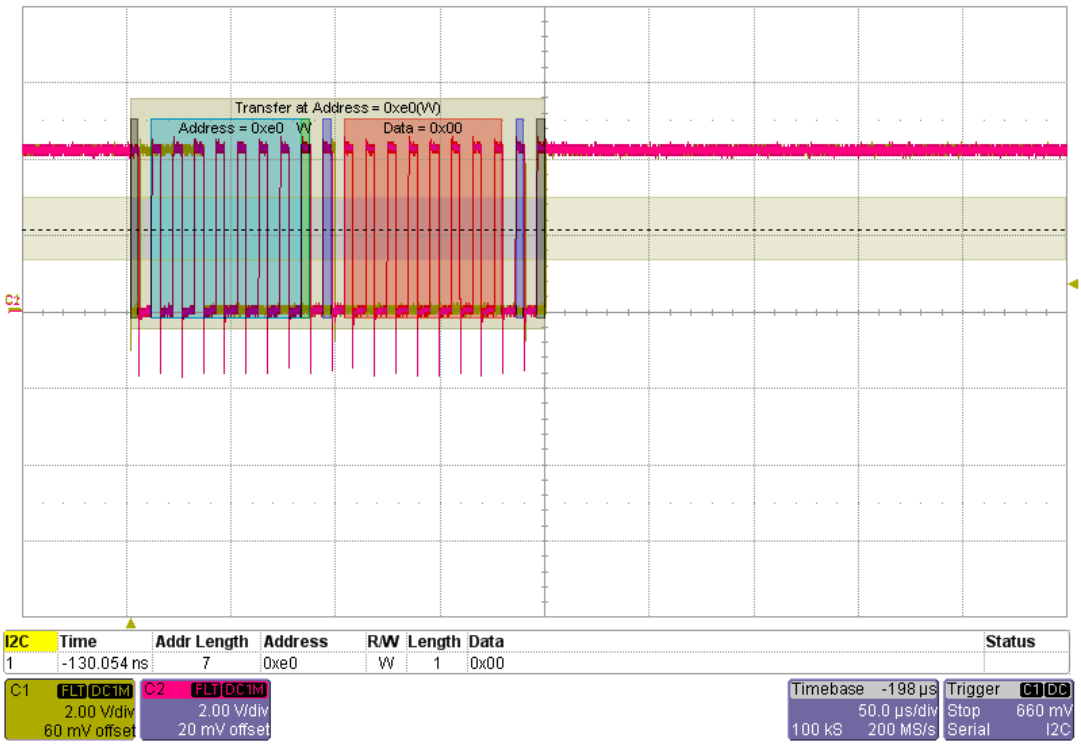


Figure 3.20: I2C transmission signals interpreted by the oscilloscope.

On the image 3.20, there is a sequence of bits, to get this image, we connected the oscilloscope to the SDA wire and observed the bit-by-bit communication.

The first bit is the start condition of the I2C communication, the next 7th is the sensor address, the 9th is the ACK (all those in blue). We find the address of the sensor on 7 bits store on 8 bits whose last bit is 0 since it is a writing (0XE0). We send the instruction 0X00 which follows the sensor address in red.

3.3 Interrupts

Now that we know how to communicate in I2C, to use a button and a LED, it is interesting in the case of the button, to use a system of interrupts in order not to allocate all the resources of the processor to the listening of the button thus allowing at any time to interrupt the current process to perform another previously defined task. To do this, there are several things to do :

- All the pins in *CubeMX* are set, so you have to tell the chip to enable the interrupt when you press the button. Go to *Categories* < *System Core* < *NVIC* and press *EXTI line[15:10] interrupts*. This command allow to create the drivers necessary to enable the interrupt from the pin 10 to the pin 15;
- We need to define when an interrupt occurs. In fact, we need to clarify what is the trigger for the interrupt. This is shown on the fig. 3.21;
- Then we need to define what actions the processor will have to perform when the interrupt occurs.

To define these two actions to the processor, they must be declared in the special file *stm32f3xx_it* called, interrupt service function file, which takes all the code relating to interrupts. To do this you must :

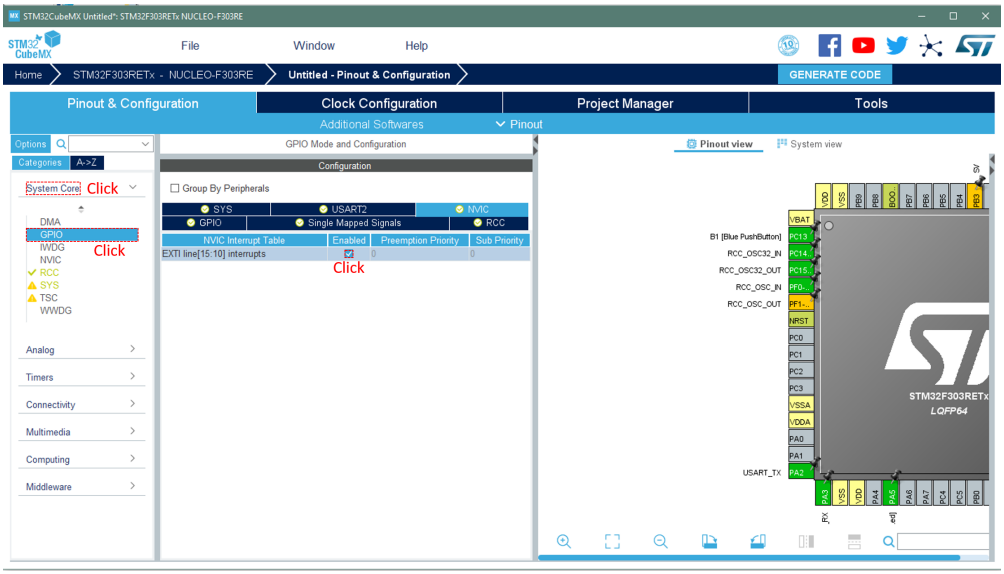


Figure 3.21: Enable the interrupts in *CubeMX* [3].

- Access the file *stm32f3xx_ it* from the **AC6** project management interface shown in fig. 3.10;
- By not touching the comments that are needed to compile it to execute the code, go to put the code put below in the indicated places;

```
/**
  * @brief This function handles EXTI line[15:10] interrupts.
  */
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  //If Interruption of the 13th pin, the led is toggled
 if(__HAL_GPIO_EXTI_GET_FLAG(GPIO_PIN_13)){
  HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
 }
  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

- Place in the main code something that completely opposes what has been put in the interrupt, in our situation, do not put anything in such a way that the LED is off. So only a delay has been placed :

```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
   HAL_Delay(1000);    // Delay of 250 ms
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

We can see in the interrupt routine, that the **__HAL_GPIO_EXTI_GET_FLAG** function is used to see when there is an event on the address corresponding to the button. If there is an event, then the LED will bling. Finally, the **HAL_GPIO_EXTI_IRQHandler** allows to clear the interrupt flag, and call the interrupt.

The only way to see that the interrupt works well is to see that each time the user presses the button, the LED that was off starts to blink.

Chapter 4

Final results

In this chapter, we will present the fully developed and functional code of our STM32 working with the distance sensor. The goal is to integrate all the knowledge previously acquired during the reading of this tutorial in order to calculate distances using the SRF02 ultrasound sensor and the possibilities offered by the button and the LED present on the STM32 chip. To do so :

- configure the different pins on *CubeMX* by setting the pin *PB8* and *PB9* to set respectively the clock and the data wires of the I2C ;
- go to the *Project Manager* section in order specify project location, the project name and the the correct compiler ;
- click on *generate code* ;
- include the necessary libraries like that :

```
/* USER CODE BEGIN Includes */
#include <stdio.h>   // It allows the UART communication with Tera Term Pro
#include <stdbool.h>  // It allows us to use printf() which is display on the Tera Term Pro
/* USER CODE END Includes */
```

- declare the necessary variables :

```
/* USER CODE BEGIN PV */


int inc =0;         // Incrementation of the button initialize to 0
int N = 4;         // Number of cases
bool flag = true;       // It allows a single display of the case
uint8_t Activation_cm[2] = {0,0x51};  // Command in the CR to get the measurements in
     centimetres (0x51), the 0 mean the address of the CR
uint8_t Activation_inches[2] = {0,0x50}; // Command in the CR to get the measurements in
     inches(0x50), the 0 mean the address of the CR
uint8_t Activation_sec[2] = {0,0x52};  // Command in the CR to get the measurements in
     microseconds(0x52), the 0 mean the address of the CR
uint8_t Capt_Point = 0x70;     // Address of the sensor SRF02
uint16_t Distance;       // 16 bits because concatenation of two 8 bits register (output)
/* USER CODE END PV */
```

- In order to use the interrupt for the button which allows the user to decide which kind if measurement he wants, we have put in the *stm32f3xx_it* file, a processus to increment the variable *inc*. So put the following code :

```
/**
  * @brief This function handles EXTI line[15:10] interrupts.
  */
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

```

```
8
9
10  //The incrementation variable is incremented in modulo N. So the sequence will be :
        0,1,2,3,0,1,2,3,0,...
11  if(__HAL_GPIO_EXTI_GET_FLAG(GPIO_PIN_13)){
12   inc = (inc+1)%N;
13
14   //The led is toggled
15   HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
16
17   //It allows a single display of the case
18   flag = true;
19
20  }
21   /* USER CODE END EXTI15_10_IRQn 0 */
22   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
23   /* USER CODE BEGIN EXTI15_10_IRQn 1 */
24
25   /* USER CODE END EXTI15_10_IRQn 1 */
26 }
```

However, we need to tell to the interrupt function that the variables *inc*, *flag* and *N* comes from the main part of the code. So declare them as extern variables in the *stm32f3xx_it* file :

```
1  /* USER CODE BEGIN Includes */
2  #include <stdbool.h>
3  /* USER CODE END Includes */
4
5  /* Private typedef -----------------------------------------------------------*/
6  /* USER CODE BEGIN TD */
7
8  /* USER CODE END TD */
9
10 /* Private define ------------------------------------------------------------*/
11 /* USER CODE BEGIN PD */
12
13 /* USER CODE END PD */
14
15 /* Private macro -------------------------------------------------------------*/
16 /* USER CODE BEGIN PM */
17
18 /* USER CODE END PM */
19
20 /* Private variables ---------------------------------------------------------*/
21 /* USER CODE BEGIN PV */
22
23       // Extern is used to link the main.c with the init_stm32f3xx_it.c
24 extern int N;    // Stil the number of cases
25 extern int inc;   // Incrementation variable
26 extern bool flag;  // Single display of the case
27 /* USER CODE END PV */
```

- As we are going to measure several types of data, we will create a *Switch case* in which we will call the same function called *Calcul_Distance*, so only the arguments will change between the different cases. Here is the function :

```
1  /* USER CODE BEGIN 4 */
2
3  int Calcul_Distance(uint8_t Capt_Point,uint8_t *Activation){
4
5     uint8_t data[1];   // Data send to the sensor
6     uint8_t MSBdata[1];  // More signficant bit
7     uint8_t LSBdata[1];  // Least signficant bit
8
9        // Write a 0 to point the Command Register (CR) and then write the command in the CR
10    HAL_I2C_Master_Transmit(&hi2c1, Capt_Point<<1 , Activation, 2, 100);
11    HAL_Delay(80);
12
13
14    //Reading of the more significant byte's sequency
15    data[0]=0x02;                // 0x02 = Address of the MSB measurement
16    HAL_I2C_Master_Transmit(&hi2c1, Capt_Point<<1, data, 1, 100);
17    HAL_I2C_Master_Receive(&hi2c1, Capt_Point<<1 , MSBdata , 1, 100);
18
19
20
21    //Reading of the least significant byte's sequency
22    data[0]=0x03;                // 0x03 = Address of the LSB measurement
```

```
23    HAL_I2C_Master_Transmit(&hi2c1, Capt_Point<<1, data, 1, 100);
24    HAL_I2C_Master_Receive(&hi2c1, Capt_Point<<1 , LSBdata , 1, 100);
25
26
27    //Concatenation of the two bytes because the measurement is on 16 bits (2bytes)
28    Distance = (int)MSBdata[0]<<8 | LSBdata[0];
29
30
31    return Distance;
32 }
33
34 /* USER CODE END 4 */
```

- Then, implement the *Switch case* :

```
1  /* Infinite loop */
2   /* USER CODE BEGIN WHILE */
3   while (1)
4   {
5    if (flag) {
6     printf("Nous sommes dans le case %i \r\n",inc);     // We are in the N case
7     flag = false;
8    }
9    switch (inc){
10
11
12
13    case 0 :
14
15      //Distance function called with a display of the output
16      Distance = Calcul_Distance(Capt_Point,Activation_cm);
17      printf("La distance vaut : %u cm\r\n",Distance);
18
19
20     //Led turned on
21      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
22      HAL_Delay(500);
23      break;
24
25    case 1 :
26
27     //Distance function called with a display of the output
28     Distance = Calcul_Distance(Capt_Point,Activation_inches);
29     printf("La distance vaut : %u pouces \r\n",Distance);
30
31     //Led turned on
32     HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
33     HAL_Delay(500);
34
35
36     break;
37
38    case 2 :
39
40     //Distance function called with a display of the output
41     Distance = Calcul_Distance(Capt_Point,Activation_sec);
42     printf("Le temps vaut : %u ms \r\n",Distance/1000);
43
44     //Led turned on
45     HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
46     HAL_Delay(500);
47
48     break;
49
50    case 3:
51
52     //Empty
53     break;
54    }
55     /* USER CODE BEGIN 3 */
56   }
57   /* USER CODE END 3 */
58 }
```

- Carry out the assembly of the card as indicated in the fig. 4.1. In order to understand the assembly, we need to refer to the ultrasonic sensor data-sheet exposed in section 1.3 where we can see the pins location to understand what the sensor pins are. Referring to the data-sheet of the chip shown in appendix B, we can find the pins on which we must connect the cables. Care must be taken not to confuse the data and clock cables for I2C communication ;

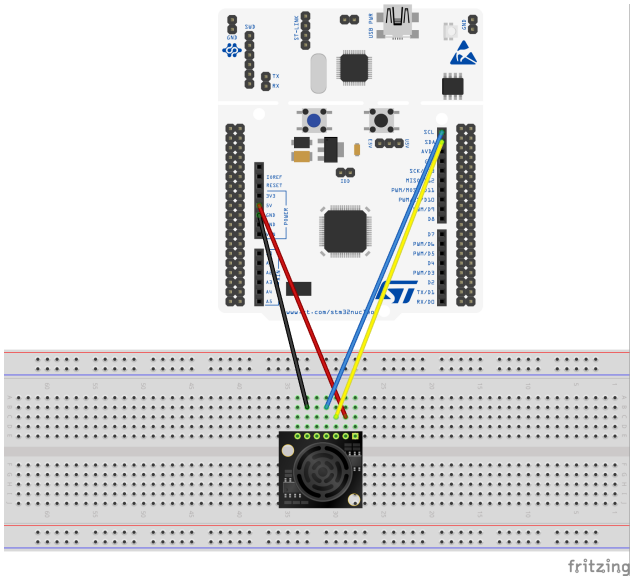


Figure 4.1: Final assembly.

- Then connect the chip to the computer using the cable making the liaison between the chip and the computer ;
- Then, referring the section 3.1.2, save, compile and place the code on the chip by clicking respectively on the different icons of the *AC6* software shown on the fig. 3.11 ;
- You can now see if the card has received the code and if everything is working correctly by opening *Tera Term Pro* ;
- If all is correct, you should see appear in the console of *Tera Term Pro*, lines similar to those placed in the fig. 4.2.

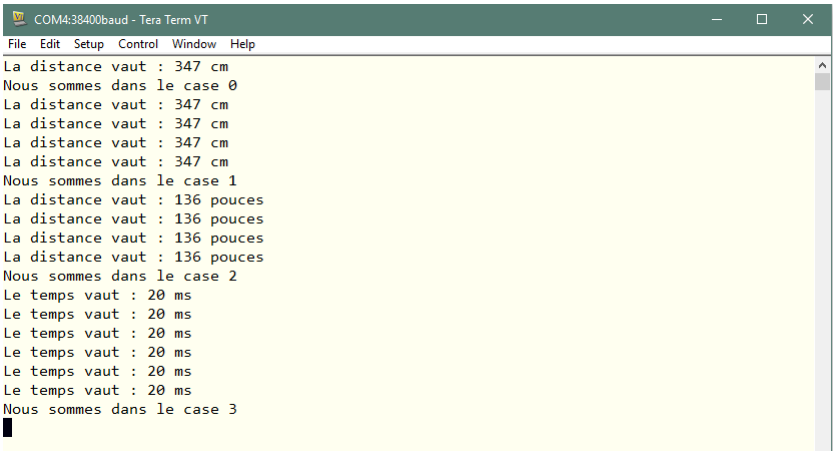


Figure 4.2: Results of the main code on *Tera Term Pro* [5].

- The implemented code can be related as the final state machine shown in fig. 4.3. The transition variable is the button that the user press each type he wants to do something different with the sensor.

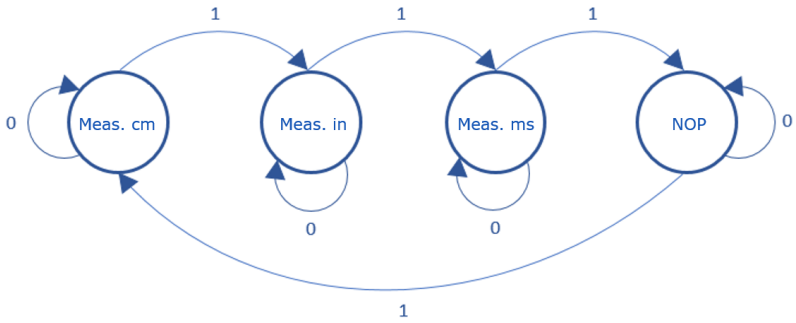


Figure 4.3: Finite State Machine

Appendices

Appendix A

Important functions

HAL_StatusTypeDef

HAL_Master_Transmit(I2C_HandleTypeDef &hi2c, uint8_t DevAddress, uint8_t *Data, uint8_t Size, uint8_t Timeout);

This function allows you to use the I2C bus to write information about the sensor. It will therefore be used to write the value of the desired command in the *Command Register*. For example, if we want to obtain the value measured in centimetres, we will have to transmit the value 0x51 in the CR.

inputs parameters are :

- &hi2c1 represents the I2C port that will be used ;
- uint8_t DevAddress«<1 is the address of the sensor;
- uint8_t *data represents the data vector that we will write, the first element of which is the address of the *Command Register* (here 0x00) ;
- uint8_t Size is the number of elements of the data vector to be taken into account;
- Timeout is a delay. When it is exceeded, it returns an error (communication failure).

Note: Before transmitting the command 0x51, it is necessary that the first element of the data vector be 0 which is actually the location of the Command Register.

HAL_Master_Receive(I2C_HandleTypeDef &hi2c, uint8_t DevAddress, uint8_t *Data, uint8_t Size, uint8_t Timeout);

This function allows you to use the I2C bus to read information at a specific sensor address. Using the example given above with the value 0x51, the measurement information is stored in register 0x02 and 0x03. These two registers are encoded on 8 bits and the distance measurement is encoded on 16 bits. It will therefore be necessary to recover and concatenate these two registers. In the *datasheet*, it is noted that register 0x02 corresponds to the least significant bits (LSB) while register 0x03 contains the most significant ones. We will see how to concatenate these in our code.

Inputs parameters are :

- &hi2c1 represents the I2C port that will be used;
- uint8_t DevAdresse«1 is the address of the sensor;

- uint8_t *data represents the address vector from which we will be read;
- Size is the number of elements of the Data vector to be read;
- Timeout is a delay. When it is exceeded, it returns an error (communication failure).

HAL_GPIO_WritePin (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)

- *GPIOx is the family of the GPIO (A,B or C);
- GPIO_Pin specifies the port bit to be written;
- PinState the data to be written on the pin.

HAL_GPIO_ReadPin(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin)

- *GPIOx is the family of the GPIO (A,B or C);
- GPIO_Pin specifies the port bit to be written;

HAL_GPIO_TogglePin(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin)

- *GPIOx is the family of the GPIO (A,B or C);
- GPIO_Pin specifies the port bit to be written;

__HAL_GPIO_EXTI_GET_FLAG

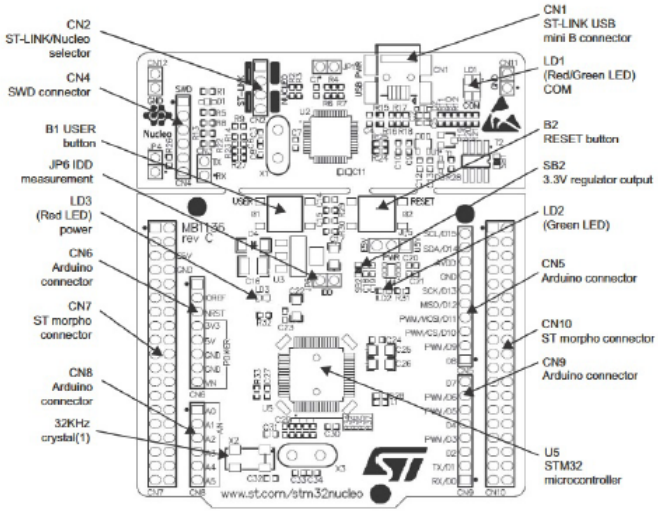
It takes the EXTI channel as an argument, where x is any number from 0 to 15. It returns true when the interrupt flag for the given channel is set. [1]

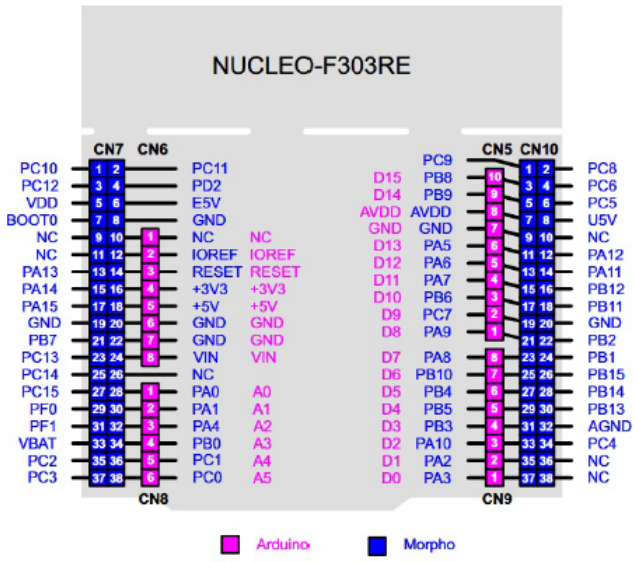
HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)

It reset the flag.

Appendix B

Location of the pins of the sensor in the data-sheet





Bibliography

[1] Camille Diou, https://www.les-electroniciens.com/sites/default/files/cours/cours_i2c.pdf. [On line ; consulted on 10-04-19].

[2] *Datasheet SRF02. Capteur de distance Ultra Sonic SRF02 en I2C – SEMI*

[3] *CubeMX* software, https://www.st.com/en/development-tools/stm32cubemx.html. [On line ; consulted on 16-04-18].

[4] *AC6* software, http://www.ac6-tools.com/content.php/content_SW4MCU/lang_en_GB.xphp. [On line ; consulted on 16-04-18].

[5] *Tera Term Pro*, software,https://ttssh2.osdn.jp/index.html.en. [On line ; consulted on 16-04-18].

[6] D. Binon, *Introduction_ Microcontroleur_ PIC.pdf*, Engeneering University of Mons.

[7] Doxygen, http://www.disca.upv.es/aperles/arm_cortex_m3/llibre/st/STM32F439xx_User_Manual/group__gpio__exported__macros.html#gaae18fc8d92ffa4df2172c78869e712fc .[On line ; consulted on 16-04-18].

[8] E. Lewiston, https://os.mbed.com/users/EricLew/code/STM32L4xx_HAL_Driver/docs/tip/group__GPIO__Exported__Functions__Group2.html#gaf5e0c89f752de5cdedcc30db068133f6 . [On line ; consulted on 17-04-18].

[9] E. Lewiston, https://os.mbed.com/users/EricLew/code/STM32L4xx_HAL_Driver/docs/tip/group__I2C__Exported__Functions__Group2.html#ga9440a306e25c7bd038cfa8619ec9a830 . [On line ; consulted on 17-04-18].