Digital Systems Electronics

Lab7-STM32

Introduction to the STM32 toolchain and NUCLEO board Experiments with Light Emitting Diodes and Switches

This document has a twofold objective. First, it provides general information about the adopted STM32 tool-chain and the Nucleo development board. The provided material is largely based on a few documents available on the web **[1] [2] [3] [4]** and two e-books **[5] [6]** . Secondly, the document describes the step-by-step development of your first STM32 project and contains some additional assignments for the Microcontroller (MCU) laboratory session 1.

1. The Tool-Chain

Before starting the development of applications for the STM32 platform, we need a complete **tool-chain**.
A tool-chain is a set of programs, compilers and tools that allows us:

- to write down our code and to navigate inside source files of our application;
- to navigate inside the application code, allowing us to inspect variables, function definitions/declarations, and so on;
- to compile the source code using a cross-platform compiler;
- to upload and debug our application on the target development board.

To accomplish these activities, we essentially need:

- an IDE (Integrated Design Environment) with integrated source editor and navigator;
- a cross-platform compiler able to compile source code for the ARM Cortex-M platform;
- a debugger that allows us to execute step by step debugging of firmware on the target board;
- a tool that allows to interact with the integrated hardware debugger of our board (the ST-LINK interface) or the dedicated programmer (e.g. a JTAG adapter).

We usually refer to term compiler as a tool able to generate machine code for the processor in our PC. A compiler is just a "language translator" from a given programming language (C in our case) to a low-level machine language, also known as assembly. A cross-platform compiler is a compiler able to generate machine code for a hardware machine different from the one we are using to develop our applications. In our case, the GCC ARM Embedded compiler generates machine code for Cortex-M processors while compiling on an x86 machine with a given OS (e.g. Windows).

The acronym IDE refers to a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, a compiler (or cross-compiler), an assembler, a linker and a debugger.

There are several complete tool-chains for the STM32 Cortex-M family, both free and commercial. IAR for Cortex-M and Keil are two of the most used commercial tool-chains for Cortex-M microcontrollers. In our case, we use STM32CubeIDE **[1]** , an advanced C/C++ development platform allowing peripheral configuration and code generation, compilation, and debug for STM32 microcontrollers and microprocessors. This IDE provides support for Windows, Linux and MacOS. It exploits Eclipse and GCC. Eclipse is an Open Source and a free Java-based IDE. It is one of the most widespread and complete development environments, available in several pre-configured versions, and customized for specific uses.
The GNU Compiler Collection (GCC) is a complete compiler suite, able to compile several programming languages to a large number of hardware architectures. GCC provides several tools to accomplish compilation tasks. These include, in addition to the compiler itself, an assembler, a linker, a debugger (known as GNU Debugger - GDB), several tools for binary files inspection, disassembly and optimization. In the ARM world, GCC is definitely the most used compiler.

STM32CubeIDE is a ready-to-use IDE, it can be download for free from the official website **[1]** together with documentation, instructions and FAQ.

IMPORTANT: Registration is necessary before accessing the available material.

2. The Nucleo board

The Nucleo family of development boards is divided in three main groups: Nucleo-32, Nucleo-64 and Nucleo-144. The name of each group comes from the MCU package type used: Nucleo-32 uses an STM32 in an LQFP-32 package; Nucleo-64 uses an LQFP-64; Nucleo-144 an LQFP-144. The Nucleo-64 group includes several different boards, each one with a given STM32 microcontroller. The available board belongs to this group and the specific version used for laboratory activities is the one known as Nucleo-F401RE **[2]** (Figure **1**).

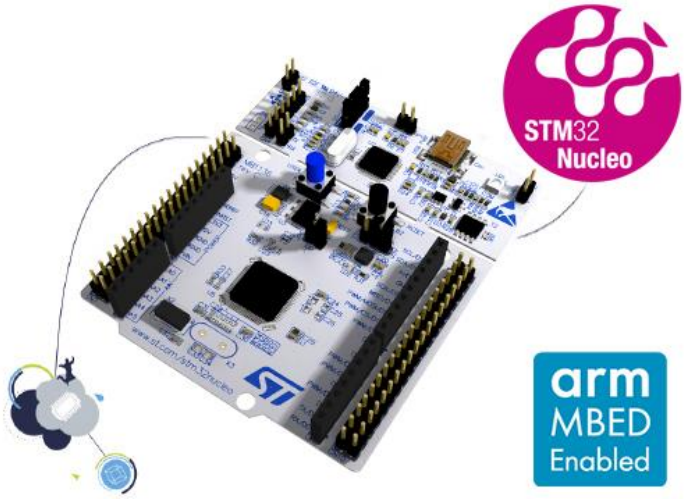


Figure 1: STM32 Nucleo board.

A summary of the boards belonging to the Nucleo family is given in Figure **2**.

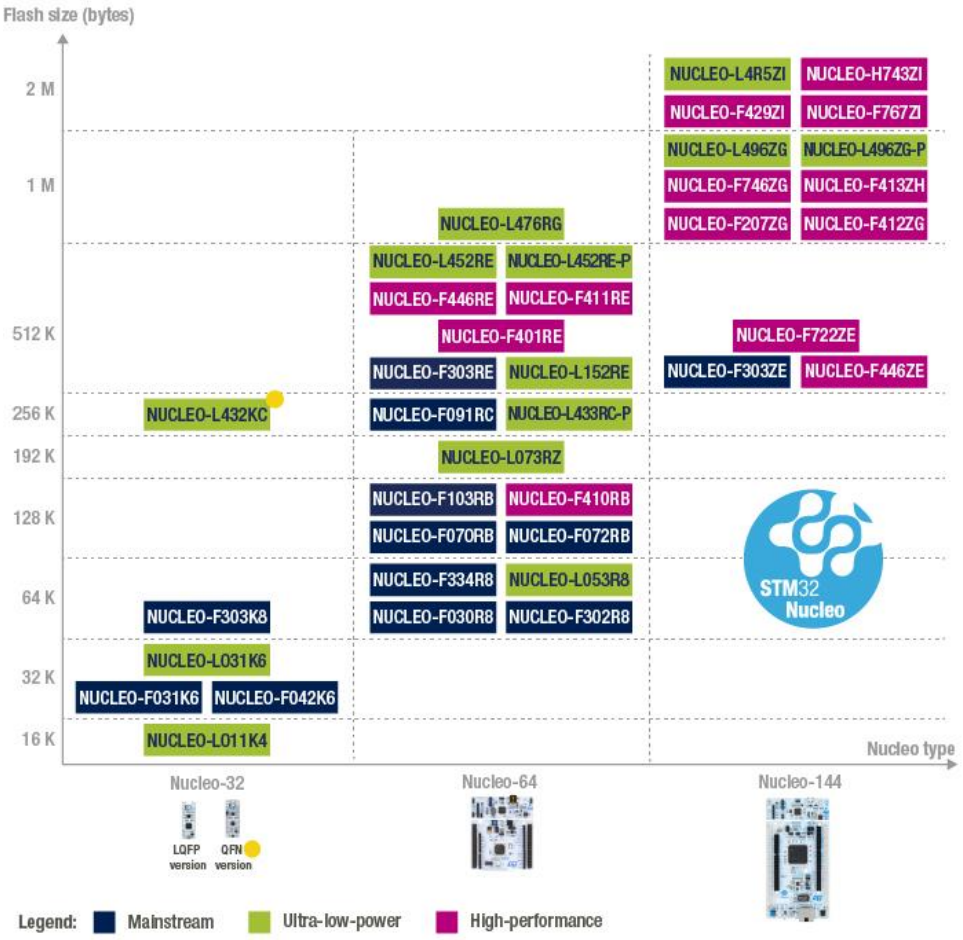


Figure 2: Nucleo boards.

The board is composed of two parts. The part with the mini-USB connector is an ST-LINK 2.1 integrated debugger, which is used to upload the firmware on the target MCU and to do step-by-step debugging. The ST-LINK interface also provides a Virtual COM Port (VCP), which can be used to exchange data and messages with the host PC. The rest of the board contains the target MCU, a RESET button, a user programmable button and an LED. The board provides pin headers to accept Arduino shields, expansion boards specifically built to expand the Arduino UNO and all other Arduino boards. Figure **3** shows the connectors supporting Arduino shields.

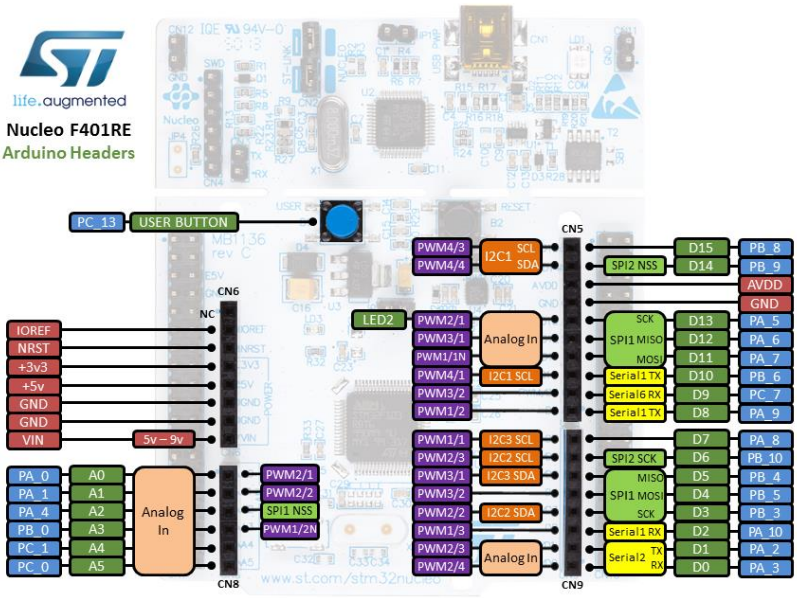


Figure 3: Arduino headers.

In addition to Arduino compatible pin headers, the Nucleo provides its own expansion connectors to access most of the MCU pins. They are two 2x19, 2.54mm spaced male pin headers, called Morpho connectors (Figure **4**).

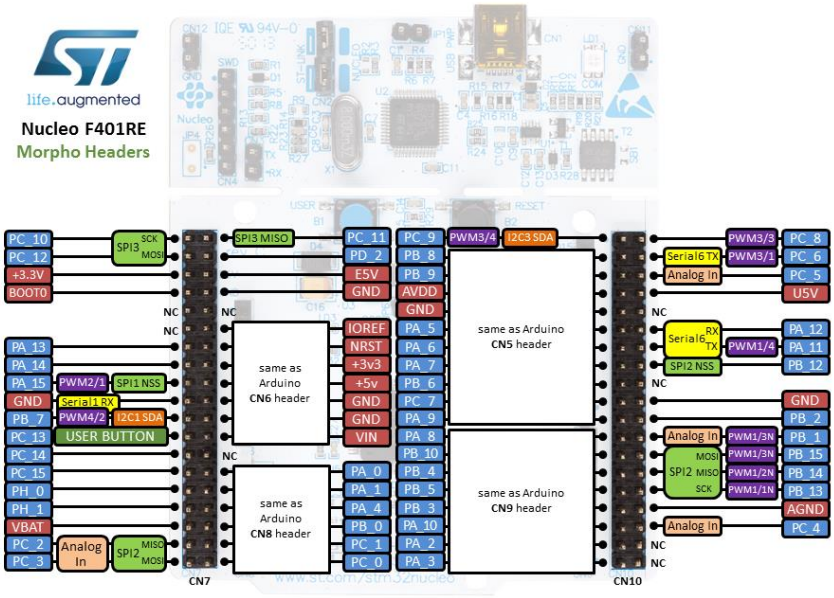


Figure 4: Morpho headers.

The top layout of the Nucleo board is shown in Figure **5**.

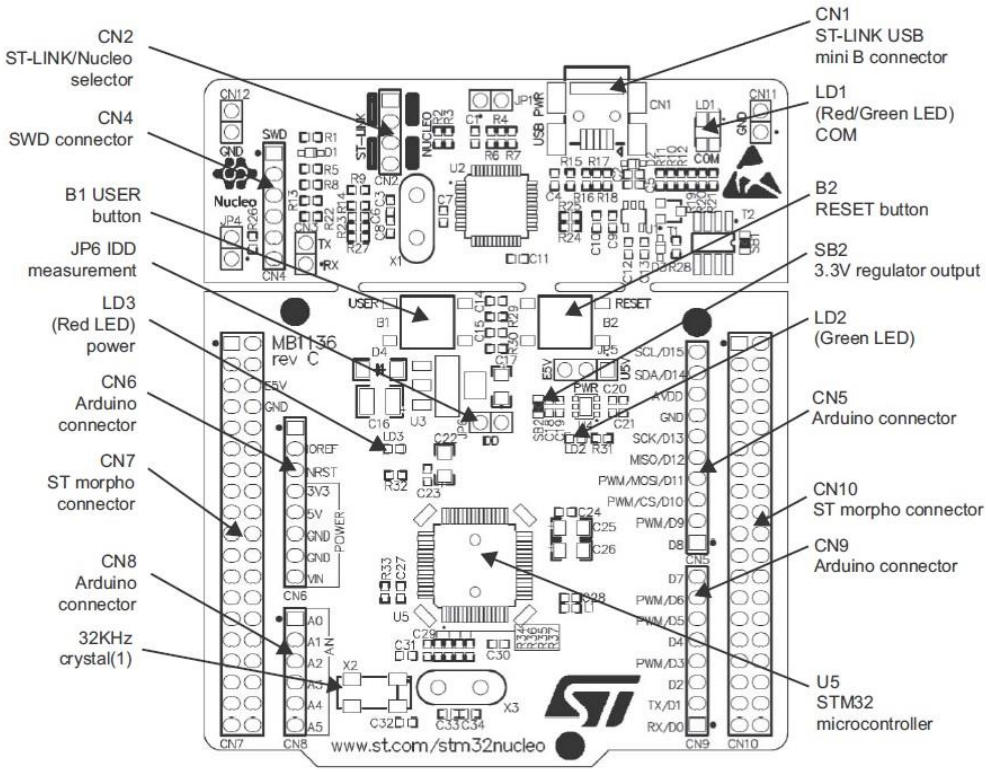


Figure 5: Top layout of the Nucleo F401RE board.

3. Starting a new project with STM32CubeIDE for STM32

As you open the STM32CubeIDE software, it will ask you to set directory of the workspace. It is the folder where we want STM32CubeIDE to store all the preferences and development artifacts. It contains different projects, which are conveniently grouped together. You can have separate workspaces for different MCUs or boards and keep there the related projects. You can easily switch between different workspaces (`File -> Switch Workspace`) and have shared resources (e.g. libraries) for different projects. You can decide to leave the default workspace as the software proposes you or change it according to your preference.
STM32CubeIDE provides libraries for the common ST development boards, including our board. When you create a new project, you can specify a board and add the related libraries, which are downloaded the first time you try to use them. By default, SW4STM32 downloads and searches for libraries in the following (hidden) folder:
`C:\Users\<UserName>\AppData\Roaming\Ac6\SW4STM32\firmwares\`.

There are multiple options to create a project:

- Use the wizard to create a new project
- Import existing project into workspace (from directory or zip)
- Copy existing project already in the workspace and rename it.

To create a new project, follow the path `File -> New -> STM32 Project`, as shown in Figure **6**.

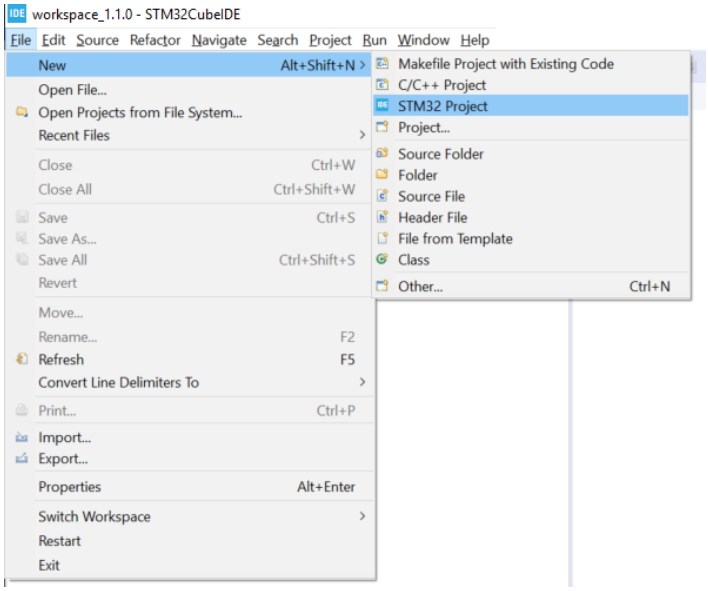


Figure 6: Create a New Project.

A new window named STM32 Project shows up, shown in Figure **7**: . It requires us to specify the Board we want to use, so that the software can compile the code correctly. Search **STM32F401RE**, as shown in **Figure 8**: . The MCU will appear on the right panel, **Figure 9**, click on it and press *Next*.

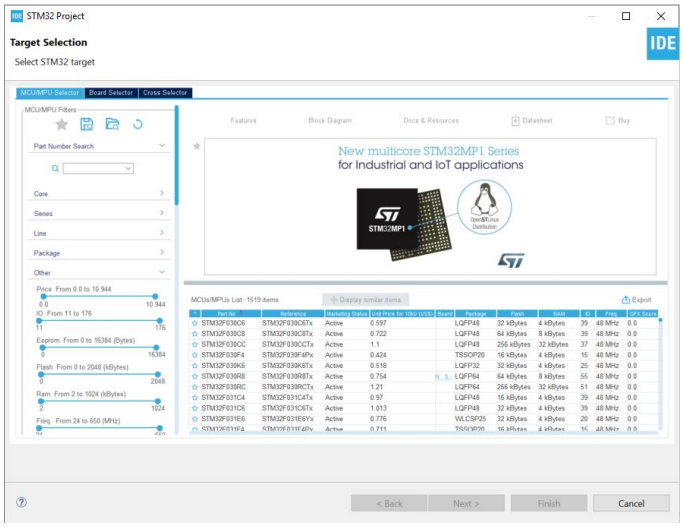


Figure 7: STM32 Project Window

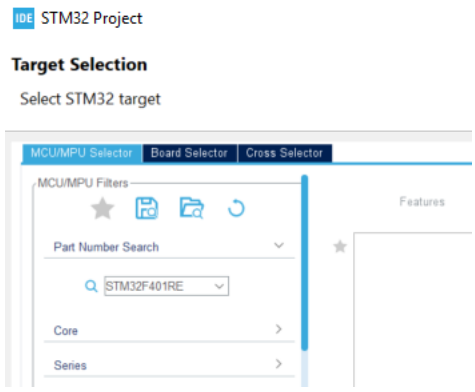


Figure 8: MCU selector Window

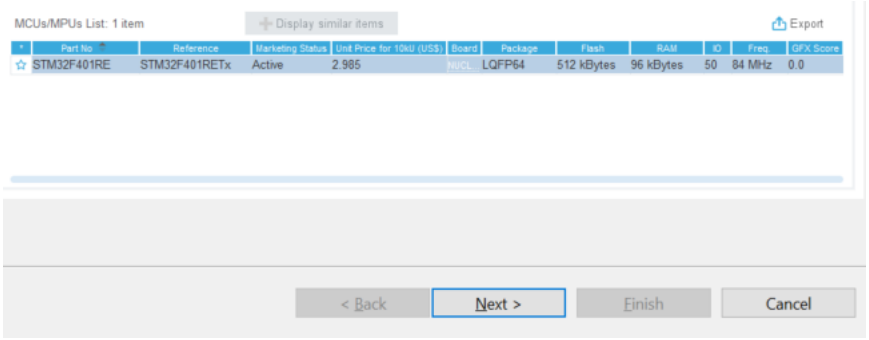


Figure 9: Selecting the MCU.

In the next panel, *Project Setup*, we have to specify the name of the project and the options of the project. Figure **10** shows the configuration we to be used for the first exercise, where *Empty* in *Targeted Project Type* is selected.

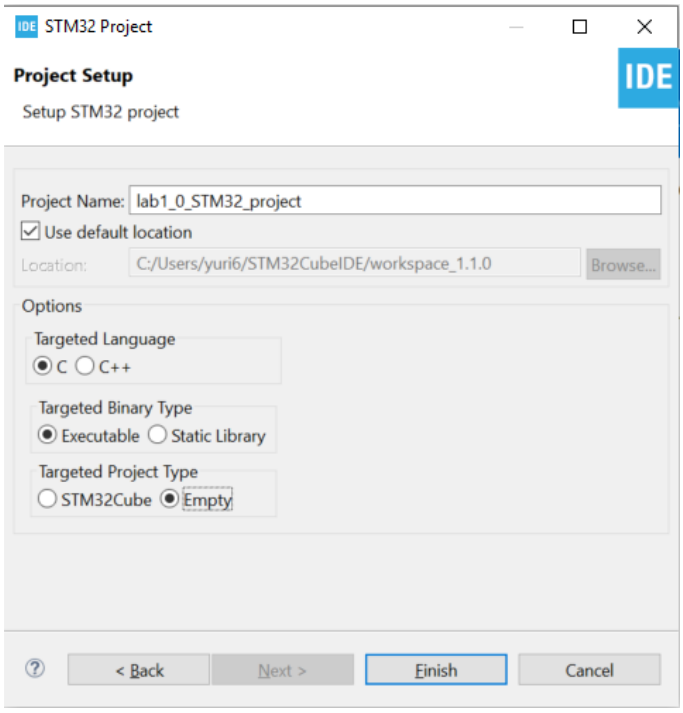


Figure 10: Selecting the project configuration.

After clicking `Finish`, the project folders are updated, and the new project appears in the *Project Explorer* panel (**Figure 11**). By expanding close to the project name, you can see the list of created directories, including:

- Core/Inc: this directory contains the Stm32f4xx_it.h Interrupt definition Header File
- Core/src: contains the project source files and specifically
  - main.c: main file, where you have to enter your code
  - syscalls.c: System calls (equivalent to stdio.h for embedded systems)
  - system_stm32f4xx.c: system initialization
- Core/startup: the very first code executed by the MCU
- Drivers/CMSIS: library developed by ARM to provide access to Cortex M CORE registers

Including src/main.c, which is the main file of our program. By default, you should see something similar to the following code, which is automatically generated by the software.

```
1  /**
2    ******************************************************************************
3    * @file    main.c
4    * @author  Auto-generated by STM32CubeIDE
5    * @version V1.0
6    * @brief   Default main function.
7    ******************************************************************************
8  */
9
10 int main(void)
11 {
       for(;;);
13 }
```

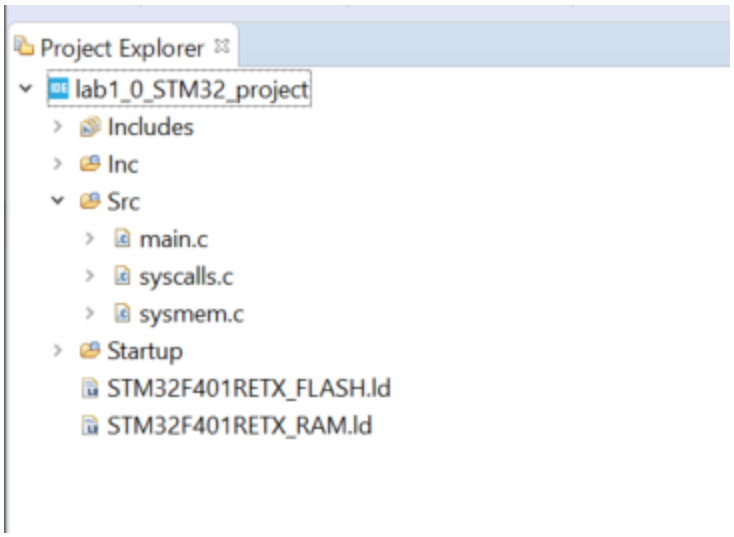


Figure 11: Created project and directories.

As an alternative to the creation of a new project, you can import an existing project and modify it. To do that, follow the command path `File -> Import`, and then select `General -> Archive file` to import a zip containing the existing project, or `General -> Existing Project into Workspace` to import the existing project from a folder.
You can also simply duplicate a project in your workspace: in the Project Explorer window, right-click on project name and select `Copy`, then right-click on Project Explorer window and select `Paste`; finally, rename the project.

4. GPIO low-level programming

In this laboratory, we use the General-Purpose I/Os (GPIO) ports. At pages 145-164 of the Reference Manual, the possible configurations with the associated circuits and all registers are reported. For this laboratory, we focus our attention on programming GPIOx_IDR (Input Data Register) and GPIOx_ODR (Output Data Register) registers for ports A and C in order to read data from input pins or to manipulate the data outputs. It is remembered that each bit of these registers is associated to a microcontroller pin that can be configured as input or output.
The Nucleo board is provided with three LEDs (see the board layout in Figure **5**) and a button.

- The **communication LD1**, a tricolor LED (green, orange, red) that provides information about ST-LINK communication status. LD1 default color is red. LD1 turns to green to indicate that communication is in progress between the PC and the ST-LINK/V2-1.
- The **user LD2** is a green LED connected to STM32 I/O PA5 (pin 21). When the I/O is HIGH value, the LED is on; when the I/O is LOW, the LED is off.
- The **B1 user BUTTON** is a pushbutton connected to STM32 I/O PC13. When the I/O is HIGH value, the BUTTON is released; when the I/O is LOW, the BUTTON is pressed.
- The **power LD3**: this red LED indicates that the STM32 part is powered and +5V power is available.

For a complete list of board connections, see the board manual, document UM1724 [3]
We will configure the STM32 I/O pin PA5 (which is part of the general purpose I/O pins, or GPIO) as an output pin. We drive it with the aim of switching LD2 on and off in a period way. We can divide the C code into two parts:

1. Register configuration
2. LED toggling

For a general understanding of the configuration and use of GPIO, see chapter 8 in **[4]** . It is important to clarify that each STM32 family (F0, F1, etc.) and each individual MCU provides its subset of peripherals, which are mapped to specific addresses. Moreover, how peripherals are implemented differs among STM32-series.
There are 6 general-purpose I/O ports. Eeach of them has four 32-bit configuration registers:

GPIOx_MODER	select the I/O direction (input, output, AF, analog)
GPIOx_OTYPER	select the output type (push-pull or open-drain)
GPIOx_OSPEEDR	select the speed (the I/O speed pins are directly connected to the corresponding GPIOx_OSPEEDR register bits whatever the I/O direction)
GPIOx_PUPDR	select the pull-up/pull-down whatever the I/O direction

Each GPIO has two 16-bit memory-mapped data registers: input and output data registers:

GPIOx_IDR	If the pin is configured as input, the register stores the associated input data. Read-only register. Use this register to read the value of an input port.
GPIOx_ODR	If the pin is configured as output, the register stores the associated output data. Use this register to output data.

Each GPIOx has a 32-bit set/reset register (GPIOx_BSRR), employed for setting/resetting each GPIOx_ODRy bit, and a 32-bit locking register (GPIOx_LCKR) for freezing the GPIO control registers by applying a specific write sequence to it.

Finally, two registers are provided to select one out of the sixteen alternate function inputs/outputs available for each I/O. With these registers, you can connect an alternate function to some other pin as required by your application. This means that a number of possible peripheral functions are multiplexed on each GPIO using two 32-bit alternate function selection registers GPIOx_AFRH and GPIOx_AFRL.

Each GPIO port bit can be individually configured by software in several modes:

- Input floating
- Input pull-up
- Input-pull-down
- Analog
- Output open-drain with pull-up or pull-down capability
- Output push-pull with pull-up or pull-down capability
- Alternate function push-pull with pull-up or pull-down capability
- Alternate function open-drain with pull-up or pull-down capability.

The architecture of the I/O port bit is shown in Figure 12.

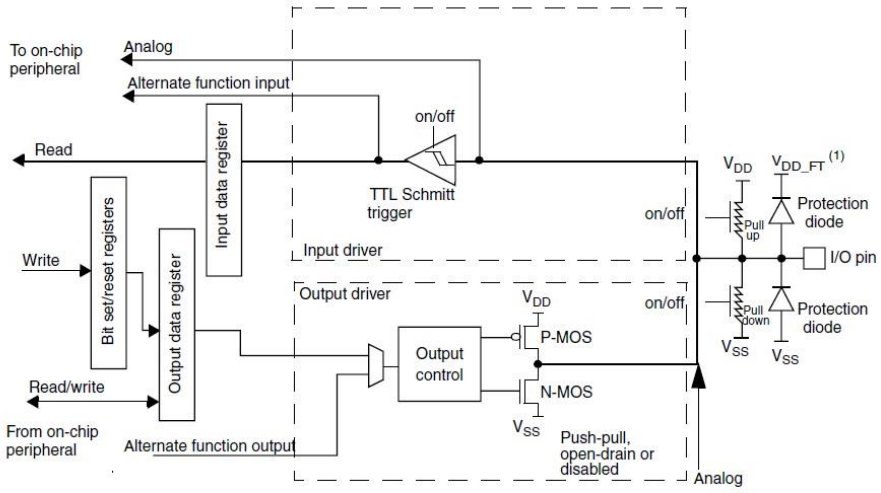


Figure 12: Architecture of the I/O port bit.

When the pin is configured as output, the value written to the output data register (**GPIOx_ODR**) is output on the I/O pin. It is possible to use the output driver in push-pull mode or open-drain mode (only the N-MOS is activated when 0 is output). The microcontroller I/O pins are connected to onboard peripherals/modules through a multiplexer that allows only one peripheral's alternate function (AF) connected to an I/O pin at a time. In this way, there can be no conflict between peripherals sharing the same I/O pin. Each I/O pin has a multiplexer with sixteen alternate function inputs (AF0 to AF15) that can be configured through the **GPIOx_AFRL** and **AFRH** registers. In addition to this flexible I/O multiplexing architecture, each peripheral has alternate functions mapped onto different I/O pins to optimize the number of peripherals available in smaller packages.

To drive the LD2, we have to configure the I/O pin PA5 as output in the GPIOA_MODER register. In general, in order to access to a register, we have to know its position in the memory. Thus we need to know the associated address. The address can be written as:

BASE_ADDRESS + **GPIOx_OFFSET** + **REGISTER_OFFSET**

The BASE_ADDRESS is associated to the peripheral: the GPIO peripheral base address is AHB1PERIPH_BASE =0x40020000. It is the same for all GPIOx registers. The GPIOx_OFFEST depends on the GPIOx whereas the REGISTER_OFFSET depends on the register we want to use.

Table 1: GPIOx and register offsets

GPIOx	GPIOx_OFFEST
GPIOA	0x0000
GPIOB	0x0400
GPIOC	0x0800
GPIOD	0x0C00
GPIOE	0x1000
GPIOH	0x1C00

Register	REGISTER_OFFSET
MODER	0x00
OTYPER	0x04
OSPEEDR	0x08
PUPDR	0x0C
IDR	0x10
ODR	0x14

Example: We want to write the address of GPIOB_ODR. The offset of GPIOB is 0x400 whereas GPIOx_ODR is 0x14. Therefore, we can write the address

GPIOB_ODR = **BASE_ADDRESS** + **GPIOx_OFFSET** + **REGISTER_OFFSET =**
= AHB1PERIPH_BASE + GPIOB_OFFSET + ODR_OFFSET =
= 0x40020000 + 0x0400 + 0x14 = 0x40020414.

We now see how to configure the LED in order for it to toggle with constant frequency.

1. Firstly, we need to set the address of the GPIOA_MODER register. This address is obtained starting from the peripheral base address (AHB1PERIPH_BASE =0x40020000) and adding the offset related to GPIOA registers (which is equal to 0) and the specific offset of MODER register (equal to 0).
```
GPIOA_MODER = (unsigned int*)0x40020000;
```
2. Similarly, we derive the address of the GPIOA_ODR register, which has offset 0x14.
```
GPIOA_ODR = (unsigned int*)(0x40020000 + 0x14);
```
3. Register GPIOA_MODER contains two bits per port bit, thus bits 11 and 10 refer to port bit 5, which is mapped to pin PA5. These bits are written by software to configure the I/O direction mode: 00 is to set the input mode (active after reset), 01 is for the general-purpose output mode, 10 is for the alternate function mode, and 11 for the analog mode. We set the output mode by means of the 0x400 mask, which allows setting the bit 10 in register GPIOA_MODER, without affecting the other ones:
```
*GPIOA_MODER = *GPIOA_MODER | 0x400;
```
4. The following instruction simply sets bit 5 in GPIOA_ODR, to have initially PA5 in the high state:
```
*GPIOA_ODR = *GPIOA_ODR | 0x20;
```

After completing the configuration, we can enter the application code. In embedded applications, the main is never left. Thus, the code always contains an infinite loop, which is executed cyclically and stopped only by a reset condition. In this case, the infinite loop is a while statement with an always true condition. Insert in the loop the code which is necessary to toggle the LED with the specified frequency. To accomplish this the loop should include two statements:

1. a toggle statements, which simply complements the value on the PA5 pin;
```
*GPIOA_ODR = *GPIOA_ODR ^ 0x20;
```
2. a delay statement, that can be implemented as an empty for loop that basically loose time counting from 0 to a predefined constant. We suggest that you define the constant with name MYWAIT and to use it as end value of the for-loop. By changing the value of the constant MYWAIT, we can modify the frequency of the LD2 blinking.
```
for (i=0; i<MYWAIT; i++){}
```

The need for repeated processing tasks separated by given amounts of delay is a very common situation in many embedded applications. Nevertheless, this approach is not a good engineering practice: while very simple, it should be avoided, because it wastes processing resources (both time and energy) only to introduce a known delay. The preferred solution makes use of timers and interrupts, as you will see in the following lab experiences.

A few additional comments on the given code example follow.

- #define MYWAIT 1000000 creates a constant named MYWAIT and sets it to the value 10^6 (this value must be long enough to have switch on and off periods in the order of seconds).
- int i; defines a local variable of type integer (4 bytes). The variable is global if the definition is outside of the main. However, global variables should be avoided as they permanently use memory.
- volatile unsigned int *GPIOA_MODER and volatile unsigned int *GPIOA_ODR defines two pointers for GPIOA_MODER and GPIOA_ODR registers. These two locations are of type unsigned int and the volatile descriptor informs the compiler that these variables should not be optimized.
- The names of the defined pointers, GPIOA_MODER and GPIOA_ODR, appear in statements that assign specific hexadecimal values to these addresses.
- Instead, by using the unary operator *, we can refer to the location corresponding to a given pointer. For example, the statement *GPIOA_ODR = *GPIOA_ODR | 0x20; calculates the bitwise OR between the content of register GPIOA_ODR and the mask 0x20, and assign the result to register GPIOA_ODR.
- The instructions

```
volatile unsigned int *RCC_AHB1ENR = (uint32_t*) (0x40023800 + 0x30);
*RCC_AHB1ENR |= 0x01U;
```

configure the clock on the AHB bus. Since the GPIOA is connected to the AHB bus, this operation is necessary to enable the GPIOA peripheral. The Reset and Clock Control (RCC) peripheral is responsible of the configuration for the whole clock tree of the MCU. In particular, the second instruction `*RCC_AHB1ENR |= 0x01U;` sets bit 0 of register to 1, therefore it enables the clock to GPIOA peripheral.

Now we are ready with the project. Open the STM32CubeIDE and create a new project, as described before. Initially, the project is empty. Open the main.c file and enter the complete code.
The STM32CubeIDE interface provides commands to compile, debug, and run the code.

- Connect the NUCLEO board to a USB port. Since an emulator of the MCU is not available, standalone code debug is not possible. We can only debug the code by connecting to the board and run the code on the real hardware.
- To compile the code, right click on the name of the project in the Project Explorer window and select *Build Project* (Figure 13). Alternatively, you can click on the hammer symbol or directly launch *Build All* from the Project menu. Compilation errors are provided in both Console and Problems windows. By clicking on an error, you can jump to the related instruction in the source file and correct it.
- To debug the code, in the Run menu, select *Debug as -> STM32 Cortex-M C/C++ Application*. This will open the Debug Interface Perspective (the software will ask you if it can open the new perspective, accept it by pressing S*witch*), see Figure 14. The Debug toolbar on the top-left corner you have all the tools used to control the execution of the code. You can execute the code one line at a time to understand where problems might exist. Breakpoints can be also inserted to automatically pause the execution in precise points of the code. The current position of the debugger is highlighted in green. You can insert breakpoints and pause code execution by right clicking on the instruction, close to the line number. On the right part of the window, you have detailed information on the MCU status, you can read the value of variables and registers to understand whether the code is correctly performing

expected operations or to directly monitor the values of registers. This allows you, for instance, to understand whether a peripheral is correctly configured.

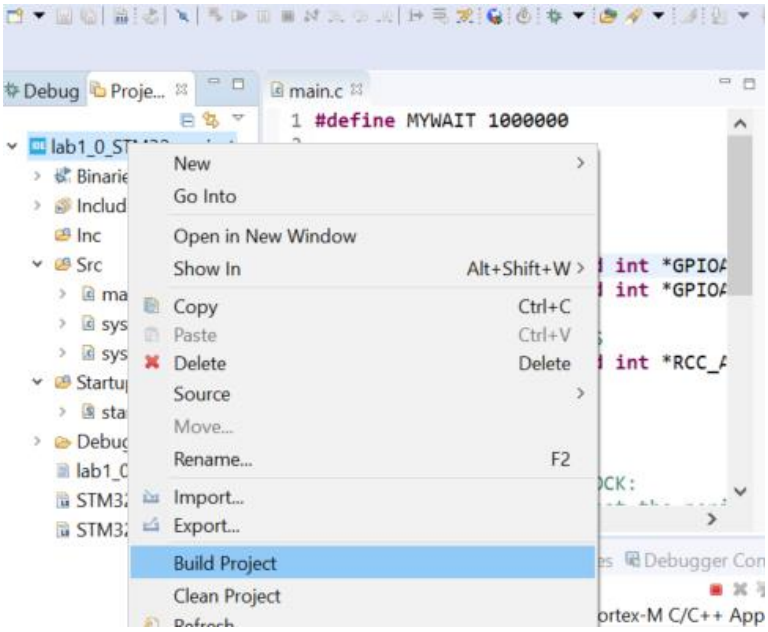


Figure 13: Build command.

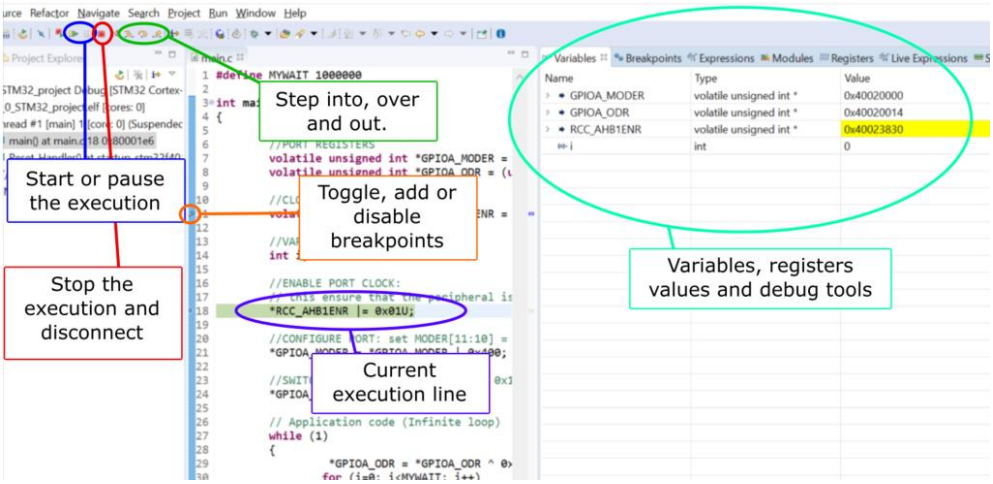


Figure 14: Debug Interface Perspective and debug commands.

In general, you can use step over, step into and step out debug commands to control the step by step code execution. The content of variables, processor registers and especially I/O registers (including GPIO registers) can be monitored on the right part of the window. At the end of the debug session, always stop the debugger, by using the *Terminate* command under the Run menu.

To modify the code so that the LD2 led is switched on while the pushbutton is pressed and switched off while the pushbutton is released, it is necessary to configure the registers associated to the pushbutton to read the status of the button. From Figure 3, it is possible to see that the pushbutton is connected to PC13, that is, bit 13 of GPIOC.

- Initialize the GPIOC peripheral so that the bit 13 is recognized as an input bit.
- Read the content of GPIOC_IDR to get the status of the pushbutton.

- Modify the code so that the LED status is switched on when the pushbutton is pressed and switched off when the pushbutton is released.

Notice that the **GPIOC will not work if you do not enable the clock on the peripheral**. To enable clock on GPIOC we need to set bit 2 of RCC_AHB1ENR to 1, as reported in the manual **[4]** (Page 117-118). Modify the line:

```
*RCC_AHB1ENR |= 0x01U;
```

so that the clock is connected both to GPIOA (setting to 1 bit 0) and to GPIOC (setting to 1 bit 2, remembering that bits count starts from 0). Leave the other bits untouched.

The registers, and the addresses, associated to GPIOC are different to the ones used for GPIOA. We remind you that the addresses of register GPIOA_ODR was obtained starting from AHB1PERIPH_BASE (0x40020000) and adding the offset related to GPIOA registers (which is equal to 0) and the specific offset of MODER register (equal to 0). For the other registers, associated to all the GPIOx peripherals, the AHB1PERIPH address is always the same, however the offsets associated to the port and to single registers are different. The offsets of GPIO peripherals, which has to be added to AHB1PERIPH_BASE to obtain the GPIOx address and the offsets for the single registers can be found on the reference manual or in the previous paragraphs, see **Table 1**: GPIOx and register offsets.

Example: the register for configuring the pull-up/pull-down mode of GPIOE will be:
AHB1PERIPH_BASE + GPIOE_OFFSET + MODER_ PUPDR = (0x40020000) + (0x1000) + (0x0C).

Notice that this is the first (and last) time you develop a low-level code for NUCLEO board, considering this it is normal that the first attempt does not succeeds. Use the debugger to understand the problem, take a look to ALL the registers you configured, see what happens to the GPIOC_IDR register when the pushbutton is pressed and/or released.

5. Starting a new project with STM32Cube

STM32Cube is a comprehensive software tool, whose goal is to significantly reduce development efforts, time and cost. It consists of (usable together or independently):

- The STM32Cube, a graphical software configuration tool that allows the generation of C initialization code using graphical wizards. It also embeds comprehensive STM32Cube MCU Packages, delivered per STM32 microcontroller Series (such as STM32CubeF4 for STM32F4 Series). These packages include the STM32Cube HAL (an STM32 abstraction layer embedded software) and the STM32Cube LL (low-layer APIs) used for the generation of configuration C code plus additional functions, such as for example power consumption calculation for a user-defined application sequence.
- STM32Cube embedded software libraries, including the handlers to simplify the porting between different STM32 devices, a collection of APIs designed for both performance and runtime efficiency, a collection of Middleware components, like RTOS, USB library, file system, TCP/IP stack, Touch sensing library or Graphic Library (depending on the MCU series).

Before proceeding, we have to make the notation clear. Indeed, the name “Cube” may be used to indicate different things, and this can lead to misunderstandings.

- **STM32CubeIDE:** this is the software we use for managing the NUCLEO board. It is used to write, compile, run and debug the code. It integrates several tools, among these, STM32Cube.
- **STM32Cube:** this is the the tool, integrated in STM32CubeIDE which allows generating the code which configures the NUCLEO board hardware.
- **STM32CubeMx:** it typically refers to a perspective of STM32CubeIDE (view mode). It’s the set of windows that allows you to use STM32Cube. Notice that in the past, two tools were necessary: “System Workbench for STM32” is the name of the software which was used to write, compile, debug and run the code, STM32CubeMX is the name of the software that allowed the generation of hardware configuration. This is the reason why the perspective is named STM32CubeMX, as STM32CubeMX (the old software) is now integrated into STM32CubeIDE.

Therefore, we use STM32Cube to generate configuration code to be imported in our application. In order to use it, we follow the same procedure we did before with a modification:

1. We create a new projecy with `File -> New -> STM32 Project`, as shown in Figure 6.
2. We specify the MCU we want to use following Figure 7: , Figure 8: , Figure 9.

Now, we slightly modify the procedure. In the previous case we wanted to create an empty project and the hardware configuration was done by hand. Here we want to use STM32Cube, therefore, we chose as *Targeted Project Type*, the option *STM32Cube*, as reported in Figure 15.

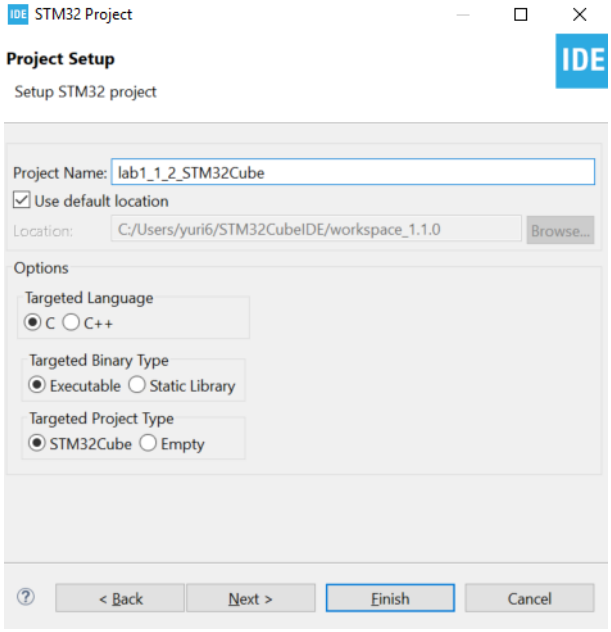


Figure 15: STM32Cube selection

A dialog will ask you about the possibility to initialize all peripherals with their default Mode, chose NO. Then, it may ask about changing the perspective to STM32CubeMx, accept it by choosing *Yes*.

The Pinout view, Figure 16, contains a representation of the MCU chip (Chip view), with the selected peripherals. In the left side, the view gives a list of all peripherals (hardware parts) and middleware libraries (software parts) that can be used with the selected MCU.
In the Chip view, pins colored in bright green are enabled. This means that CubeMX will generate the needed code to configure that pin according to its functionalities. A pin is colored in grey when the corresponding peripheral is not enabled, so no setup code will be automatically generated. Yellow pins are power source pins, and their configuration cannot be changed. BOOT and RESET pins are colored in khaki, and their configuration cannot be changed. By moving the mouse pointer over an MCU pin, we can read where the pin is mapped.
In order to enable one pin, we click on it and then we can configure it. The configurations of pins PA5 (corresponding to user LD2) and PC13 (corresponding to user button) as GPIO_Output and GPIO_Input respectively are reported in Figure 17.

Figure 16: Main STMCube window.

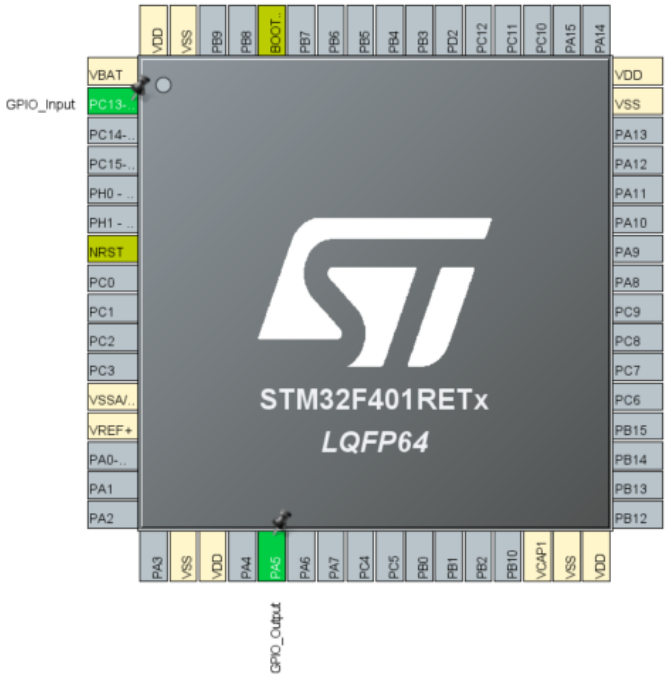


Figure 17: Main STMCube window, with PA5 and PC13 pins enabled.

By expanding one of the elements on the left another list appears. These list shows by means of icons and colors which peripherals are available and enabled or disabled. In the clock view, we can set both the main core and the peripherals clocks. All clock sources and PLLs configurations are presented in a graphical way. Configuration view allows to select several options for peripherals and software components. Configuration options defined in this view impact the automatically generated C source code. Finally, the Tools tab embeds the Power Consumption Calculator (PCC) view is a feature of STMCube that, given

a microcontroller, a battery model and a user-defined power sequence, provides an estimation of the average power consumption and battery life.

Once STMCube has created the new project, move to the *Project Manager* tab (**Errore. L'origine riferimento non è stata trovata.**). Leave everything as default, then move to *Advanced Settings* to configure the LL drivers **for each peripheral**, as in Figure **19**. In the Code Generator tab, select the options of Figure **20**. Finally, we can generate the C initialization code, with the command

Project > Generate Code

The generated code is available in the proper folder of the Eclipse workspace. Several subfolders are generated. In particular, *Inc* and *Src* folders contain headers and source files of the skeleton application generated by STMCube, and the STM32xxxx_HAL_Driver folder (that has this name even if you work with LL drivers) inside the Drivers one, is the whole ST LL for the selected microcontroller series.

The GPIO configuration code generated by STM32Cube is already inserted in the main code and name definitions are already imported. We only need to write the user code.

Notice that the code generated by CubeMX is organized in different sections defined by particular comments. You have to write your code in the dedicated section. Otherwise your code will be deleted with successive generation by STM32Cube. In the following an example:

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
INSERT YOUR INCLUDES HERE
/* USER CODE END Includes */
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
INSERT YOUR DEFINES HERE
/* USER CODE END PD */
```

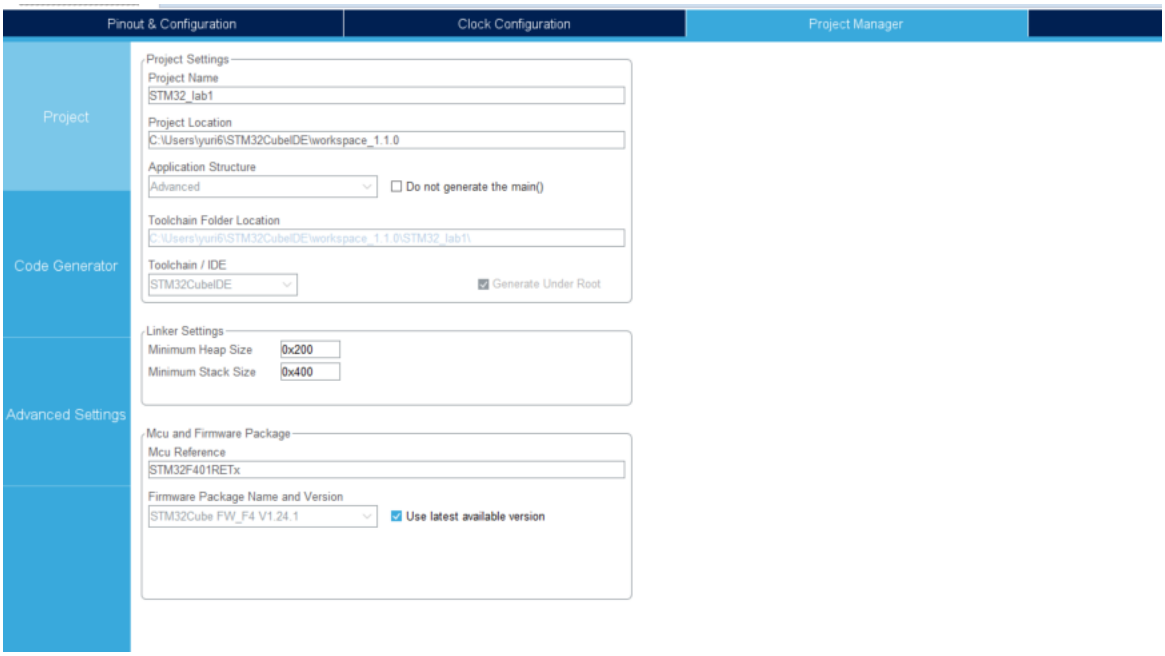


Figure 18: Project Settings window.

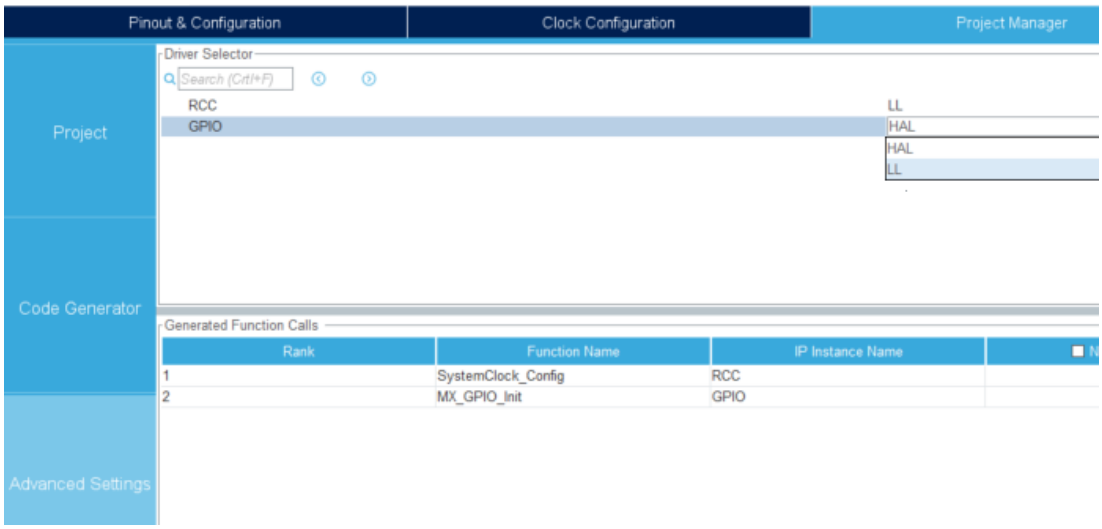


Figure 19: Advanced Settings window.

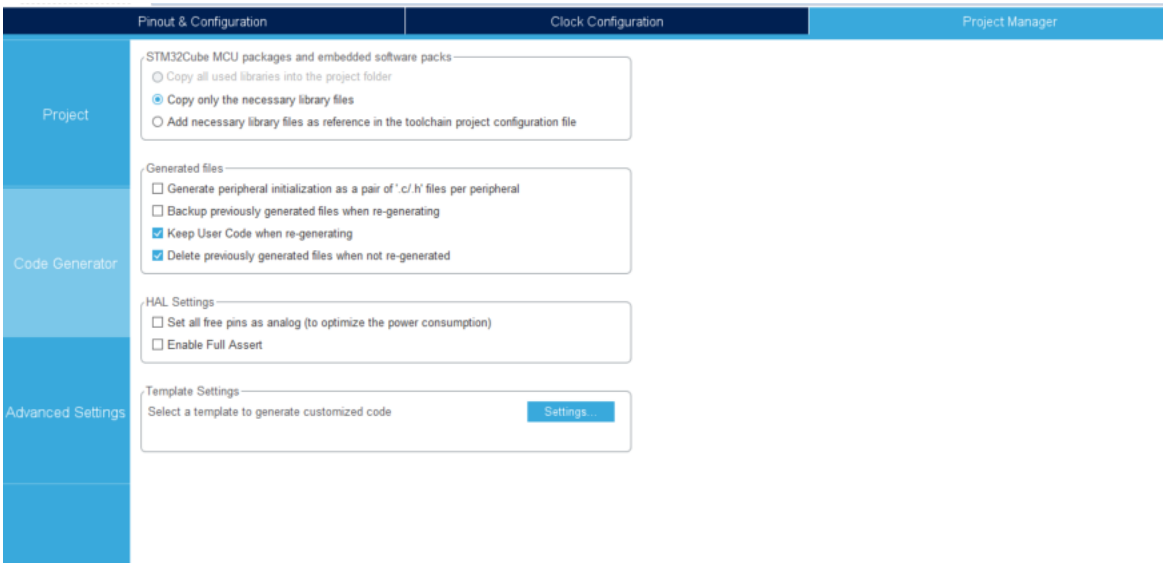


Figure 20: Project Settings window, Code Generator tab.

6. GPIO programming using Low-Layer (LL) paradigm

Low-Layer programming provides a set of functions that can be employed in order to manipulate the content of these registers. In particular, we use two macros that are described at pg. 1286 of the User Manual:

- `LL_GPIO_ReadReg(__INSTANCE__,__REG__)`, for reading a value in a GPIO register. Two parameters are required:
  - `__INSTANCE__`: GPIO Instance
  - `__REG__`: Register to be read

The macro returns the value of the register.

- `LL_GPIO_WriteReg(__INSTANCE__,__REG__,__VALUE__)`, for writing a value in a GPIO register. Three parameters are required:
  - __INSTANCE__: GPIO Instance
  - __REG__: Register to be written
  - __VALUE__: Value to be written in the register

  The macro does not return any value.

The parameters __INSTANCE__ and __REG__ are properly two memory locations: the first one is the base address for all those containing the values of the configuration and data registers of a specific port, while __REG__ is an offset with the respect to the previous one; the addition of __INSTANCE__ and __REG__ values provides the memory address containing the value of the register of interest. In order to avoid the research of addresses on the datasheet, it is possible to find the __INSTANCE__ and __REG__ addresses already defined in the header file \Drivers\CMSIS\Device\ST\STM32F4xx\Include\stm32f401xe.h . It is sufficient to specify the port name as __INSTANCE__ and the register of interest as __REG__.

For example, if we want to configure an open-drain output on pin D11 by setting the corresponding bit to 1 in the OTYPER register (see pg. 157 on the datasheet), without modifying the other bits, we must write in our code

```
LL_GPIO_WriteReg( GPIOD, OTYPER,(LL_GPIO_ReadReg(GPIOD,OTYPER) | 0x800) );
```

so that the bit OT11 is set to 1 through a bitwise OR operation (see the masks' reference for other cases).
The most important registers for GPIO programming (see pg. 157-164 of the Reference Manual for all registers) are:

- GPIO port mode register (GPIOx_MODER), to configure the I/O direction mode. 00: Input (reset state), 01: General purpose output mode, 10: Alternate function mode, 11: Analog mode).
- GPIO port output type register (GPIOx_OTYPER). These bits are written by software to configure the output type of the I/O port. 0: Output push-pull (reset state) and 1: Output open-drain.
- GPIO port pull-up/pull-down register (GPIOx_PUPDR). These bits are written by software to configure the I/O pull-up or pull-down 00: No pull-up/pull-down, 01: Pull-up, 10: Pull-down, 11: Reserved.
- GPIO port input data register (GPIOx_IDR). They contain the input value of the corresponding I/O port.
- GPIO port output data register (GPIOx_ODR). They contain the output value of the corresponding I/O port.

It is remembered that we must program the GPIOx_IDR and GPIOx_ODR only, while the others are automatically configured by CubeMX.

7. Plotting signals in the debugger

In STM32CubeIDE it is possible to visualize the signals on the board pins using a debugger feature called *SWV Data Trace Timeline Graph*. In order to do it, some steps are needed.
First of all, one needs to use a **global** variable in which to save the value of the pin to be plotted. For example, if one wants to plot the value of the LED2 pin of the board, the following variable needs to be defined.

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

static uint16_t led=0;

/* USER CODE END 0 */
```

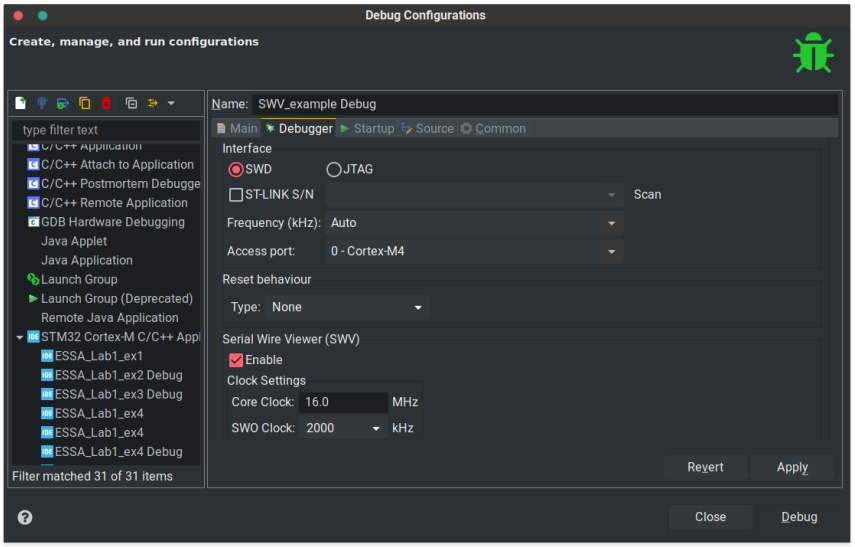
One can notice how the variable is defined as a **static** one. This is because it will be used only to read the LED register. Hence, if it is was not a static one, the compiler would remove it to optimize the code, since it is an unused variable.

Then, one needs to insert a `HAL_Delay()` function in the loop, otherwise the plot debugger would not work. In our case, we generate a periodic signal on the LED through `HAL_Delay()`. The presence of this function is **fundamental**! Even if it is not used in your original code, include it (for example, as `HAL_Delay(0)` so that it does not affect your code performances).

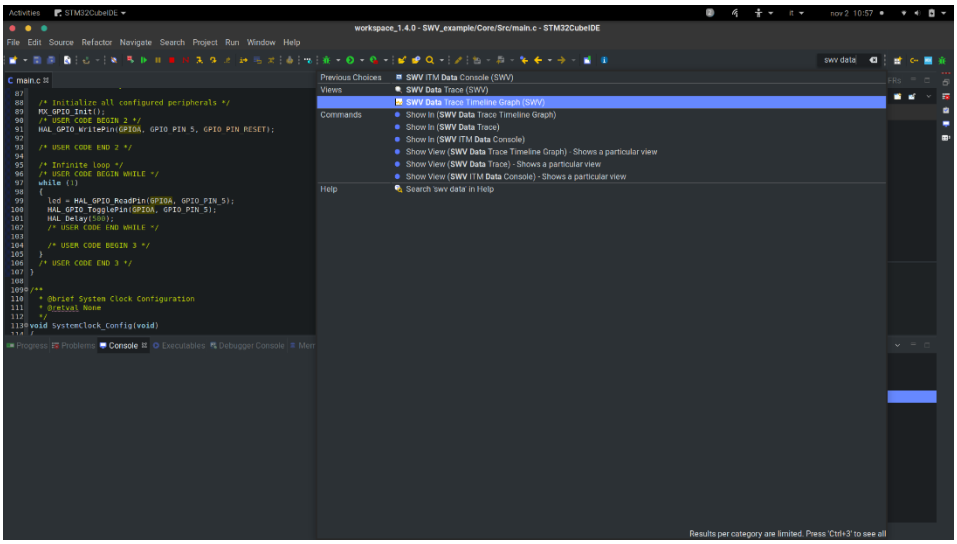
```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    led = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_5);
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
    HAL_Delay(500);
    /* USER CODE END WHILE */

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

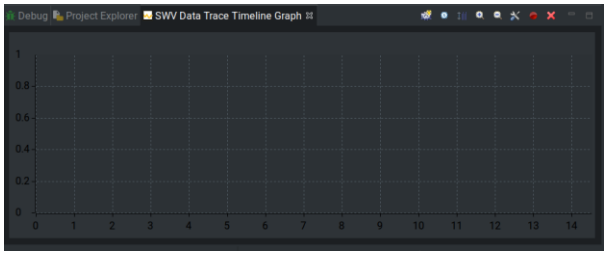
One can notice how the LED status is saved inside `led` each time, in order to keep the variable updated. For this reason, this statement has to be inserted in the loop, otherwise it would not be possible to track the pin value while the code is running.
Then, one starts the debugging phase. When debugging, one need to select a specific option inside the menu *Debug configurations*.



In the *Debugger* section, one need to enable the *SWD* interface and to enable the *Serial Wire Viewer (SWV)*. Then, one can click *Apply* and *Debug*.
Once the debugging mode is entered, one needs to add the SVW viewer. This is done by searching for *SWV Data Trace Timeline* in the *Quick search* menu on the top right.



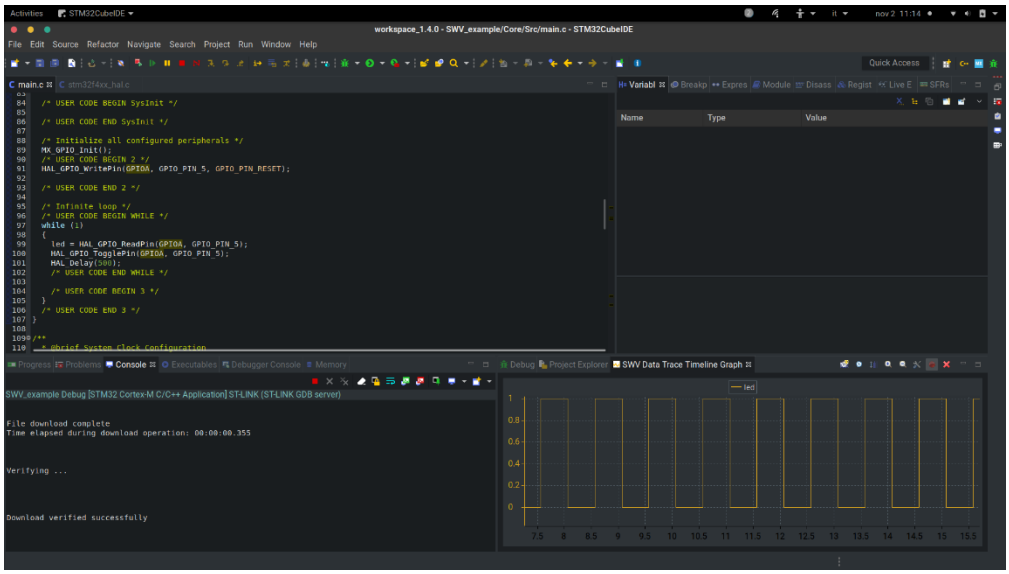
Once this is done, an *SVW Data Trace Timeline Graph* tab will appear on the bottom left. One clicks on it and the viewer appears.



Then, the waveform needs to be configured. One needs to click on the “wrench and screwdriver” icon and a configuration menu appears.

NB: il freq clock deve essere quello scelto durante la configurazione del codice

Four waveforms can be plotted at the same time. To each waveform it corresponds a comparator. In order to plot a variable, one needs to add the variable name in the *Var/Addr* field and to enable the comparator as shown in the figure above, and to select *Data Value + PC* in the *Generate* menu. Then, one can click OK.
After this, one needs to start the trace recording, so that the variable value is recorded during code execution. This is done by clicking on the red button in the *SVW Data Trace Timeline Graph* window. After this, one can start the debugging phase, by clicking on the *Resume* button (the green right-oriented arrow one) on the top bar of the menu. The result should look like this.



References

[1] https://www.st.com/en/development-tools/stm32cubeide.html
[2] http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-nucleo/nucleo-f401re.html#design-scroll
[3] UM1724 User manual (STM32 Nucleo-64 board), Nov. 2016 (description of the Nucleo board).
[4] RM0368 Reference manual, May 2015 (guide to the use of memory and peripherals in the STM32).
[5] Agus Kurniawan , "Getting Started With STM32 Nucleo Development", 1st Edition, ISBN 9781329075559, 2015
[6] Carmine Noviello, "Mastering STM32", 2017, available for sale at http://leanpub.com/mastering-stm32
[7] http://www.st.com/en/ecosystems/stm32cube.html?querycriteria=productId=SC2004