Button Read using LL libraries

rev1.0 24/03/2020

GOAL

Read the state of the button on the board and debounce it

PREREQUISITES

Software needed:

- STM32IDE

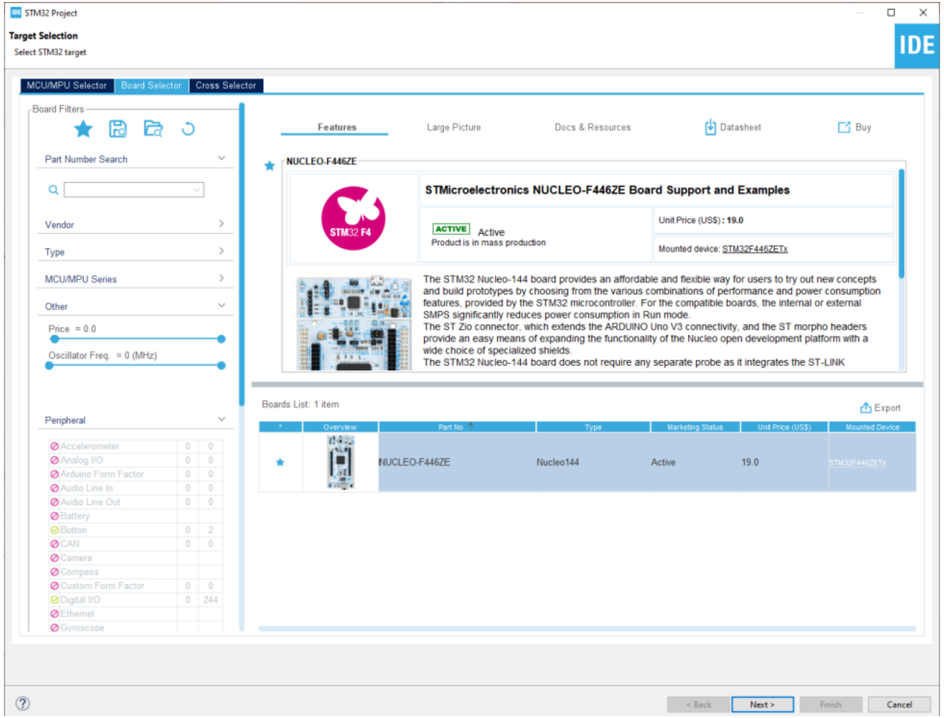
Hardware used in this example:

- **NUCLEO-F446ZE**

Start a new project

From the stm32IDE software click on File -> New -> STM32 Project.

Select your board or your uC and click *next.*



Target Selection

Select STM32 target

Price = 0.0

Oscillator Freq. = 0 (MHz)

Accelerometer	0	0
Analog I/O	0	0
Arduino Form Factor	0	0
Audio Line In	0	0
Audio Line Out	0	0
Battery		
Button	0	2
CAN	0	0
Camera		
Compass		
Custom Form Factor	0	0
Digital I/O	0	244
Ethernet		
Gyroscope		

NUCLEO-F446ZE

STMicroelectronics NUCLEO-F446ZE Board Support and Examples

ACTIVE Active
Product is in mass production

Unit Price (US$) : **19.0**

Mounted device: STM32F446ZETx

The STM32 Nucleo-144 board provides an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 microcontroller. For the compatible boards, the internal or external SMPS significantly reduces power consumption in Run mode.
The ST Zio connector, which extends the ARDUINO Uno V3 connectivity, and the ST morpho headers provide an easy means of expanding the functionality of the Nucleo open development platform with a wide choice of specialized shields.
The STM32 Nucleo-144 board does not require any separate probe as it integrates the ST-LINK

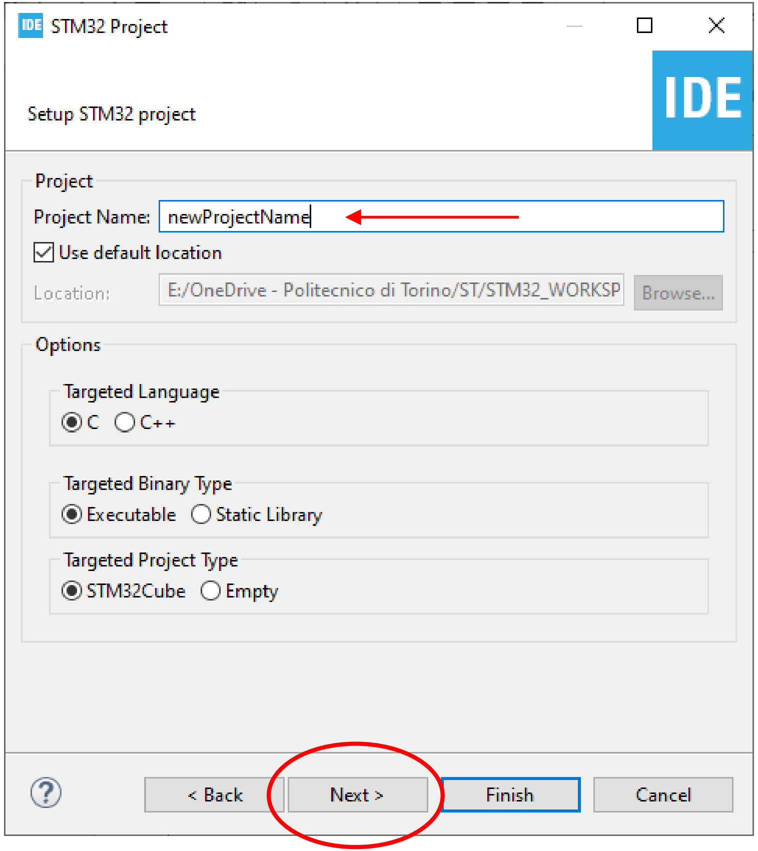
Boards List: 1 item

*	Overview	Part No	Type	Marketing Status	Unit Price (US$)	Mounted Device
★		NUCLEO-F446ZE	Nucleo144	Active	19.0	STM32F446ZETx

Start a new project

Type the name of your project and click next.

By default the project will be created in the workspace folder.



STM32 Project

Setup STM32 project

Project

Project Name: newProjectName

☑ Use default location

Location: E:/OneDrive - Politecnico di Torino/ST/STM32_WORKSP

Options

Targeted Language

● C ○ C++

Targeted Binary Type

● Executable ○ Static Library

Targeted Project Type

● STM32Cube ○ Empty

Start a new project

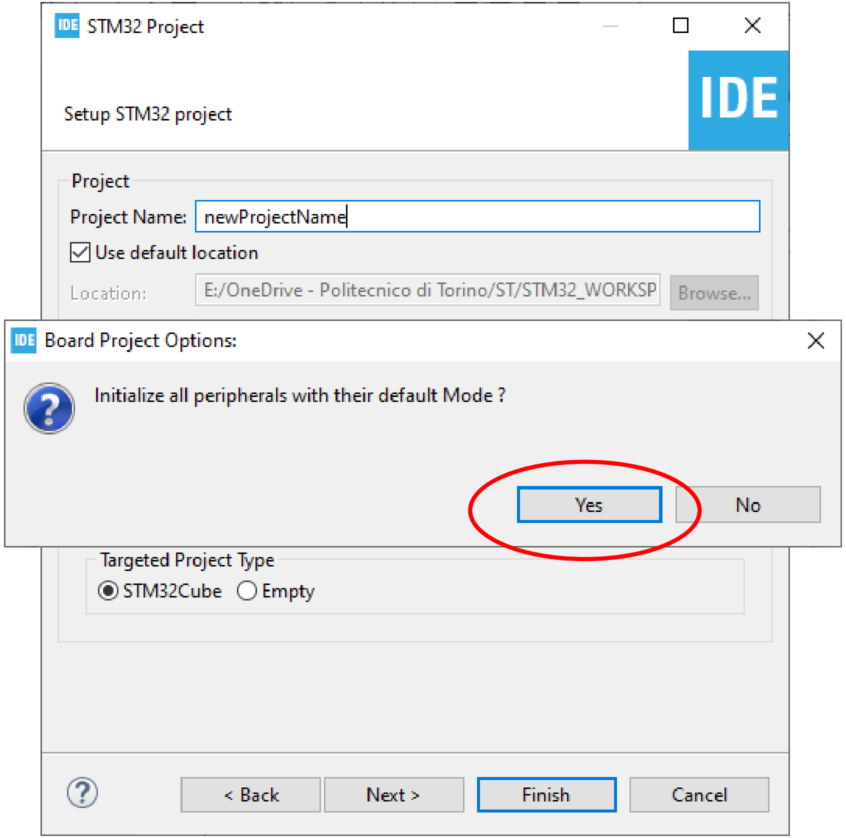
Type the name of your project and click next.

By default the project will be created in the *workspace* folder.

The *STM32IDE* has the option to initialize all the peripheral with their ***default*** mode:

Clicking *Yes* the *USART3*, all the *LEDs* and the blue *UserButton* will be configured as default.

Click *Yes*.



STM32 Project

Setup STM32 project

Project

Project Name: newProjectName

☑ Use default location

Location: E:/OneDrive - Politecnico di Torino/ST/STM32_WORKSP

Board Project Options:

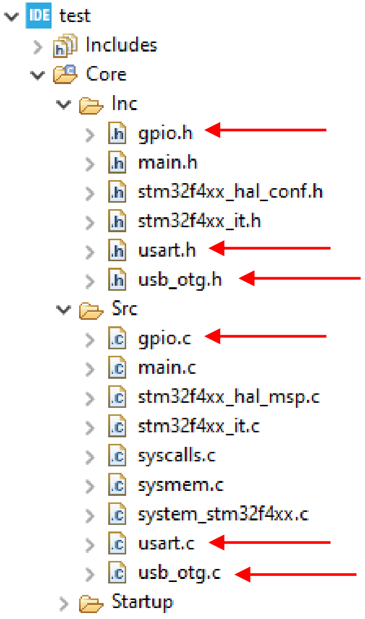
Initialize all peripherals with their default Mode ?

Targeted Project Type

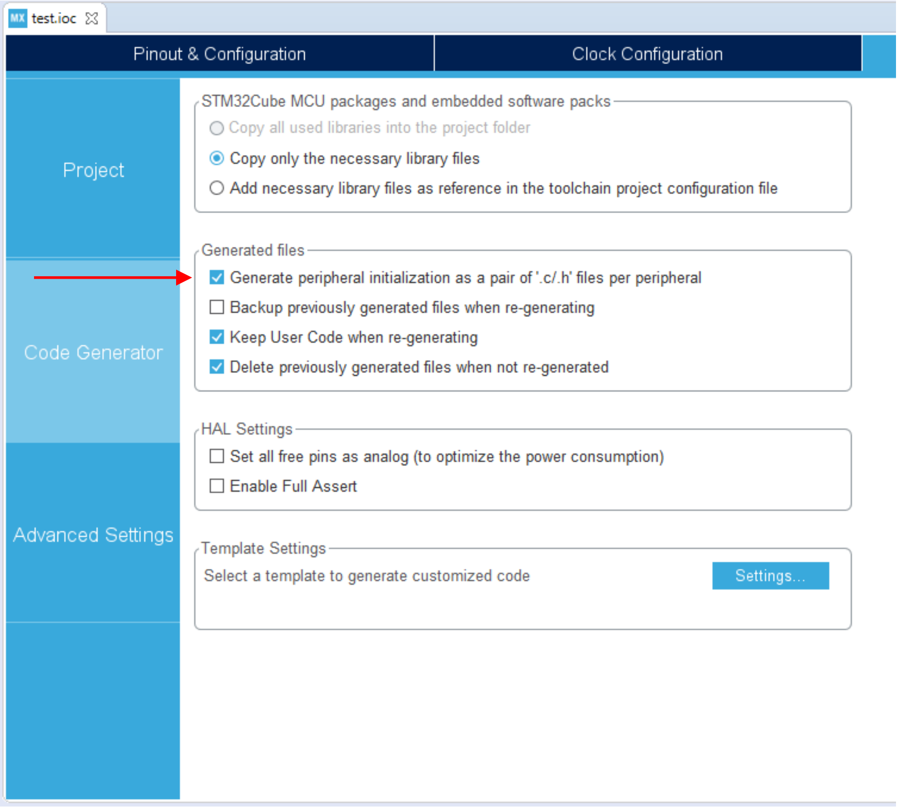
● STM32Cube ○ Empty

Project Manager

In the Code Generator Tab check the ***Generate peripheral initialization […]*** box: each periperhal will have a disting *periph.c* and *periph.h* files.



```
test
  Includes
  Core
    Inc
      gpio.h
      main.h
      stm32f4xx_hal_conf.h
      stm32f4xx_it.h
      usart.h
      usb_otg.h
    Src
      gpio.c
      main.c
      stm32f4xx_hal_msp.c
      stm32f4xx_it.c
      syscalls.c
      sysmem.c
      system_stm32f4xx.c
      usart.c
      usb_otg.c
  Startup
```



STM32Cube MCU packages and embedded software packs

○ Copy all used libraries into the project folder

● Copy only the necessary library files

○ Add necessary library files as reference in the toolchain project configuration file

Generated files

☑ Generate peripheral initialization as a pair of '.c/.h' files per peripheral

☐ Backup previously generated files when re-generating

☑ Keep User Code when re-generating

☑ Delete previously generated files when not re-generated

HAL Settings

☐ Set all free pins as analog (to optimize the power consumption)

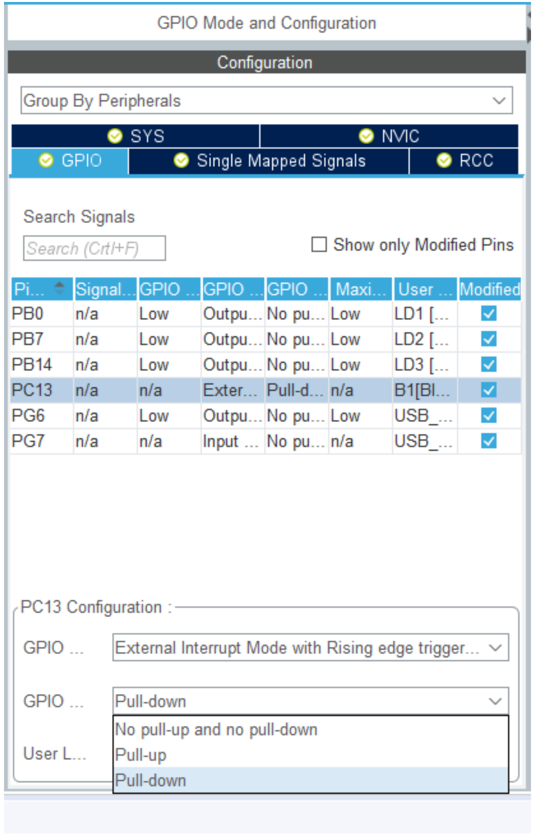
☐ Enable Full Assert

Template Settings

Select a template to generate customized code

Pull-up o Pull-down ?

- After the last overview of the *GPIOs*, it should be noted that sometimes, especially for the *INPUT* pins, it is convenient to decide whether these should be ***pull-up*** or ***pull-down***, how to choose?



GPIO Mode and Configuration

Configuration

Group By Peripherals

Pi...	Signal...	GPIO ...	GPIO ...	GPIO ...	Maxi...	User ...	Modified
PB0	n/a	Low	Outpu...	No pu...	Low	LD1 [...	☑
PB7	n/a	Low	Outpu...	No pu...	Low	LD2 [...	☑
PB14	n/a	Low	Outpu...	No pu...	Low	LD3 [...	☑
PC13	n/a	n/a	Exter...	Pull-d...	n/a	B1[Bl...	☑
PG6	n/a	Low	Outpu...	No pu...	Low	USB_...	☑
PG7	n/a	n/a	Input ...	No pu...	n/a	USB_...	☑

PC13 Configuration :

GPIO ... External Interrupt Mode with Rising edge trigger...

GPIO ... Pull-down

User L...

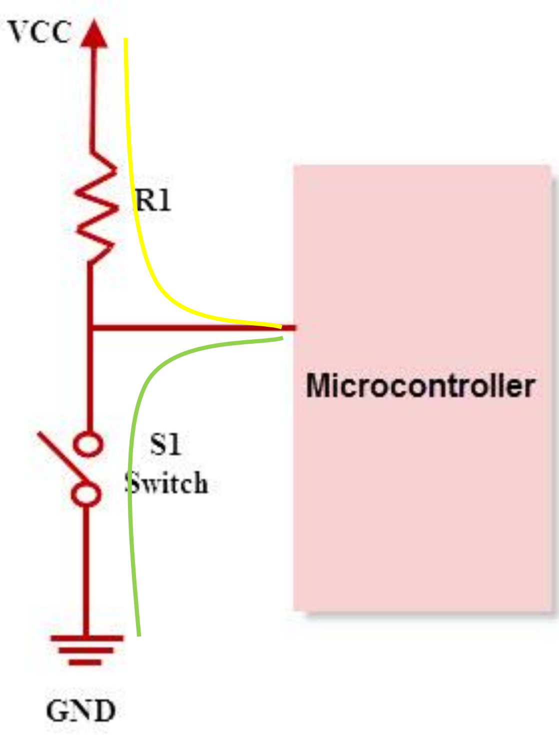
Pull-Up

In a digital circuit, the pull-up configuration is useful, when the switch that we are going to read is connected to GND:

- *OPEN* → VCC
- *CLOSED* → GND

In this way the input read will be:

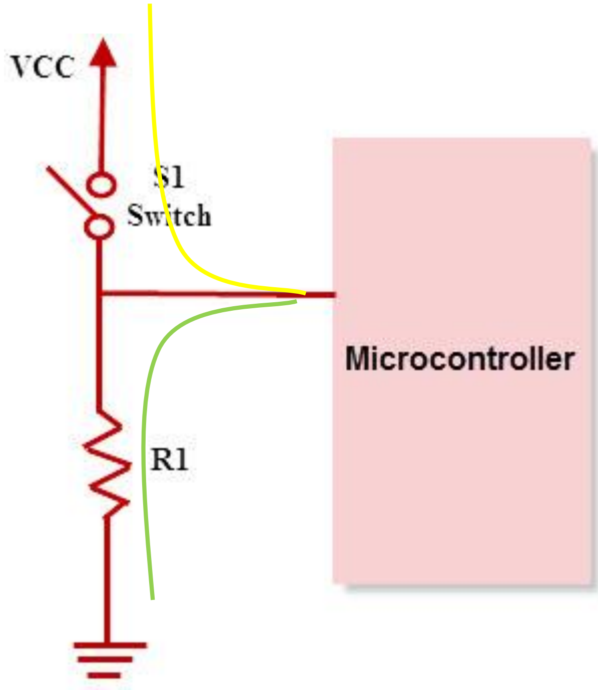
- ‘1’ → Open Switch
- ‘0’ → Closed switch



Pull-Down

n a digital circuit, the pull-down configuration is useful, when the switch we are going to read is connected to VCC:

- *CLOSED* → VCC
- *OPEN* → GND

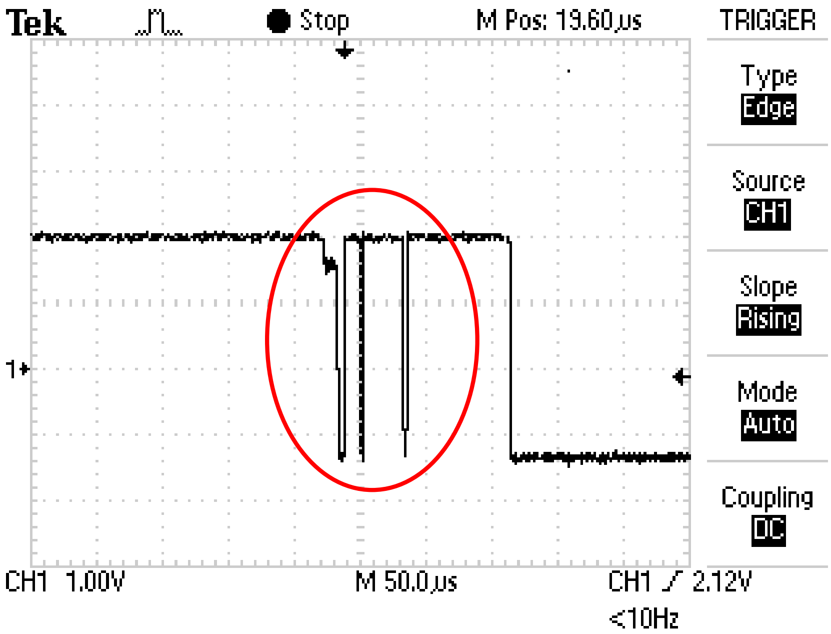


In this way the input read will be:

- ‘1’ → Switch Closed
- '0' → Open switch

Debouncing, what is it?

- When reading an input, in particular a button, it is often useful to ***debounce*** it, that is, to prevent the input noise from causing spurious switching. How to do ? There are various ways, both hardware and software, here we will focus on the second type.



Tek Stop M Pos: 19.60µs

TRIGGER

Type Edge

Source CH1

Slope Rising

Mode Auto

Coupling DC

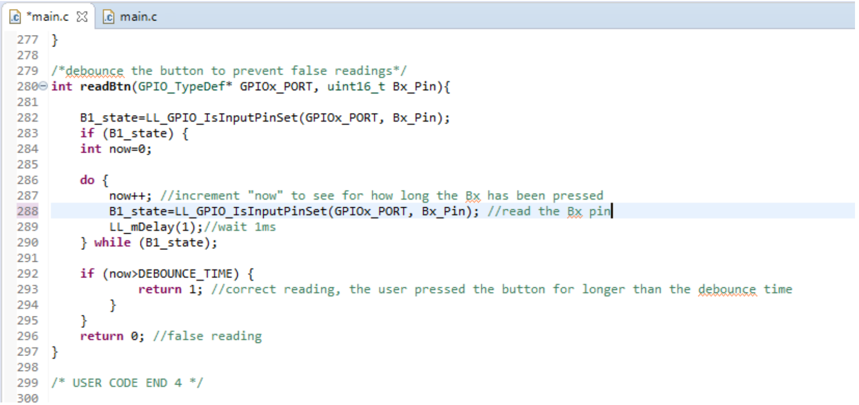
CH1 1.00V M 50.0µs CH1 ⁄ 2.12V

<10Hz

Switch Bouncing prodotto dalla pressione di un pulsante

Software debouce

- An example of software debounce can be the one shown here:
- Through the function ***LL_GPIO_IsInputPinSet()*** the status of the input is checked: in the event that this assumes the *'SET'* value then a counter is started which has the purpose of controlling how long the input is kept at a high level. The delay of 1ms causes the value assumed by the *now* variable to correspond to the time, in ms, during which our button was pressed.

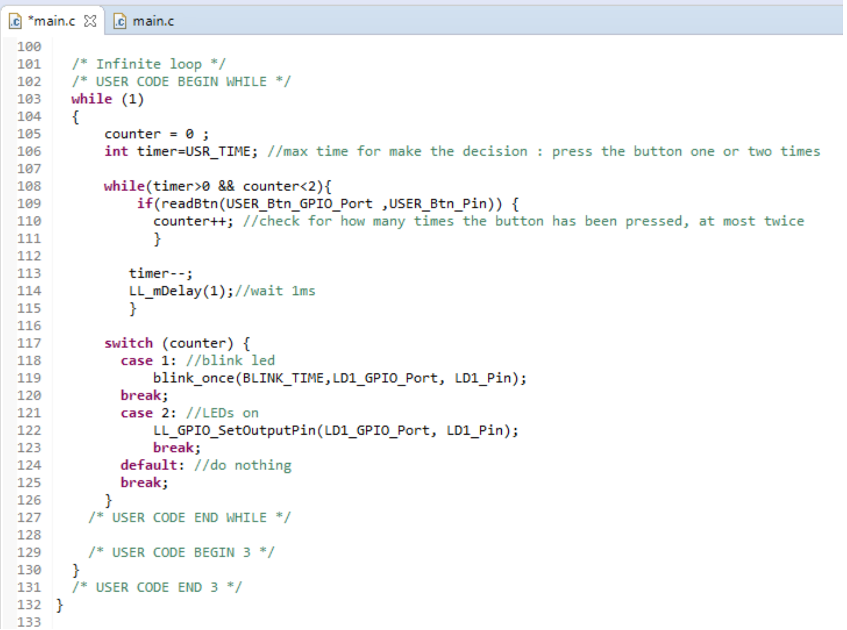


```c
}

/*debounce the button to prevent false readings*/
int readBtn(GPIO_TypeDef* GPIOx_PORT, uint16_t Bx_Pin){

    B1_state=LL_GPIO_IsInputPinSet(GPIOx_PORT, Bx_Pin);
    if (B1_state) {
    int now=0;

    do {
        now++; //increment "now" to see for how long the Bx has been pressed
        B1_state=LL_GPIO_IsInputPinSet(GPIOx_PORT, Bx_Pin); //read the Bx pin
        LL_mDelay(1);//wait 1ms
    } while (B1_state);

    if (now>DEBOUNCE_TIME) {
            return 1; //correct reading, the user pressed the button for longer than the debounce time
        }
    }
    return 0; //false reading
}

/* USER CODE END 4 */

```

- Consideriamo una pressione volontaria del bottone se questo è stato attivo continuamente per un tempo maggiore al **DEBOUNCE_TIME**, altrimenti viene considerata come spuria e quindi rumore.

How many times has the button been pressed?

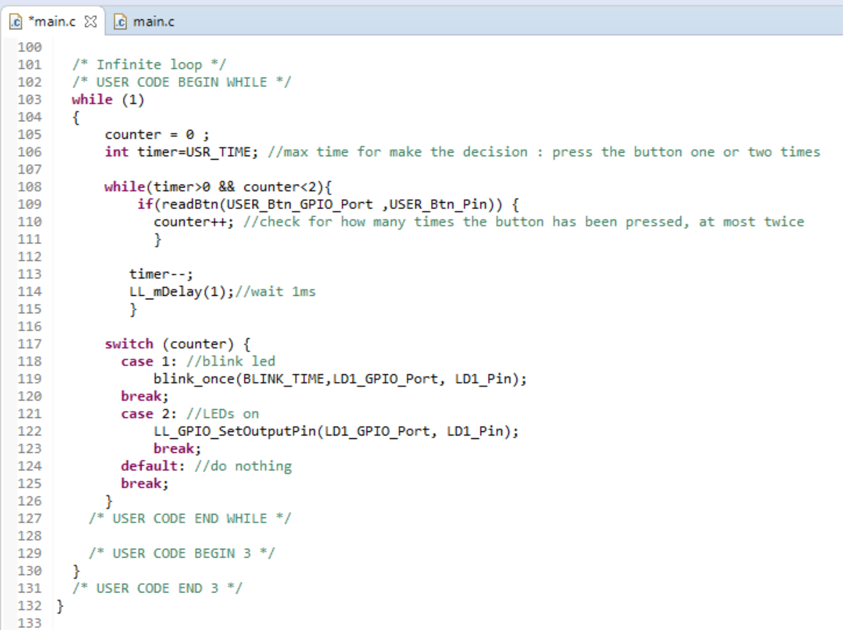
- It may be useful to understand how many times the button has been pressed and then evaluate the different cases, for example:
  - 1 press → Blink
  - 2 presses → LED on
- One strategy could be to define a USR_TIME time limit in which the user can press the button once or twice.
- During this first phase, the counter variable takes into account how many times the switch is pressed.



```c

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      counter = 0 ;
      int timer=USR_TIME; //max time for make the decision : press the button one or two times

      while(timer>0 && counter<2){
          if(readBtn(USER_Btn_GPIO_Port ,USER_Btn_Pin)) {
            counter++; //check for how many times the button has been pressed, at most twice
            }

         timer--;
         LL_mDelay(1);//wait 1ms
         }

      switch (counter) {
        case 1: //blink led
            blink_once(BLINK_TIME,LD1_GPIO_Port, LD1_Pin);
        break;
        case 2: //LEDs on
            LL_GPIO_SetOutputPin(LD1_GPIO_Port, LD1_Pin);
            break;
        default: //do nothing
        break;
      }
    /* USER CODE END WHILE */

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

```

How many times has the button been pressed?

- As in the example above, the 1ms delay at the end of this pre-evaluation phase makes USR_TIME the ms that the user has available to make his choice.
- At the end of this phase it is possible to evaluate how many times the button has been pressed using the value contained in the counter variable.

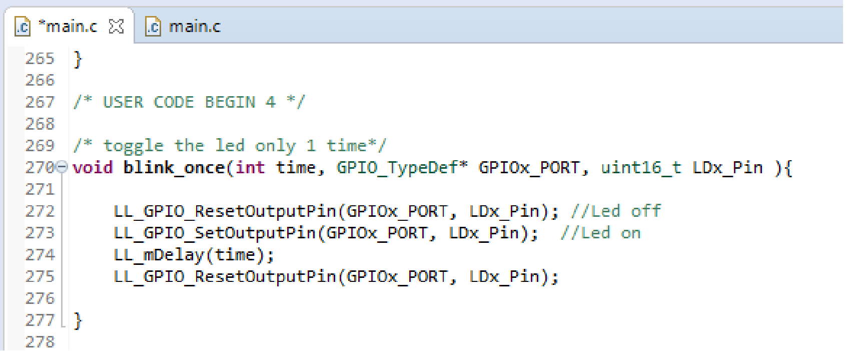


```c

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      counter = 0 ;
      int timer=USR_TIME; //max time for make the decision : press the button one or two times

      while(timer>0 && counter<2){
          if(readBtn(USER_Btn_GPIO_Port ,USER_Btn_Pin)) {
            counter++; //check for how many times the button has been pressed, at most twice
            }

         timer--;
         LL_mDelay(1);//wait 1ms
         }

      switch (counter) {
        case 1: //blink led
            blink_once(BLINK_TIME,LD1_GPIO_Port, LD1_Pin);
        break;
        case 2: //LEDs on
            LL_GPIO_SetOutputPin(LD1_GPIO_Port, LD1_Pin);
            break;
        default: //do nothing
        break;
      }
    /* USER CODE END WHILE */

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

```

blink_once()

- For completeness, the function that allows the LED to flash only once is also shown.

- The code is very simple to interpret but it is important to understand the ***RESET→SET*** passage on the first two lines which allows you to be sure of obtaining a flash **ON / OFF** and not vice versa.



```c
}

/* USER CODE BEGIN 4 */

/* toggle the led only 1 time*/
void blink_once(int time, GPIO_TypeDef* GPIOx_PORT, uint16_t LDx_Pin ){

    LL_GPIO_ResetOutputPin(GPIOx_PORT, LDx_Pin); //Led off
    LL_GPIO_SetOutputPin(GPIOx_PORT, LDx_Pin);  //Led on
    LL_mDelay(time);
    LL_GPIO_ResetOutputPin(GPIOx_PORT, LDx_Pin);

}

```