PERLATECNICA

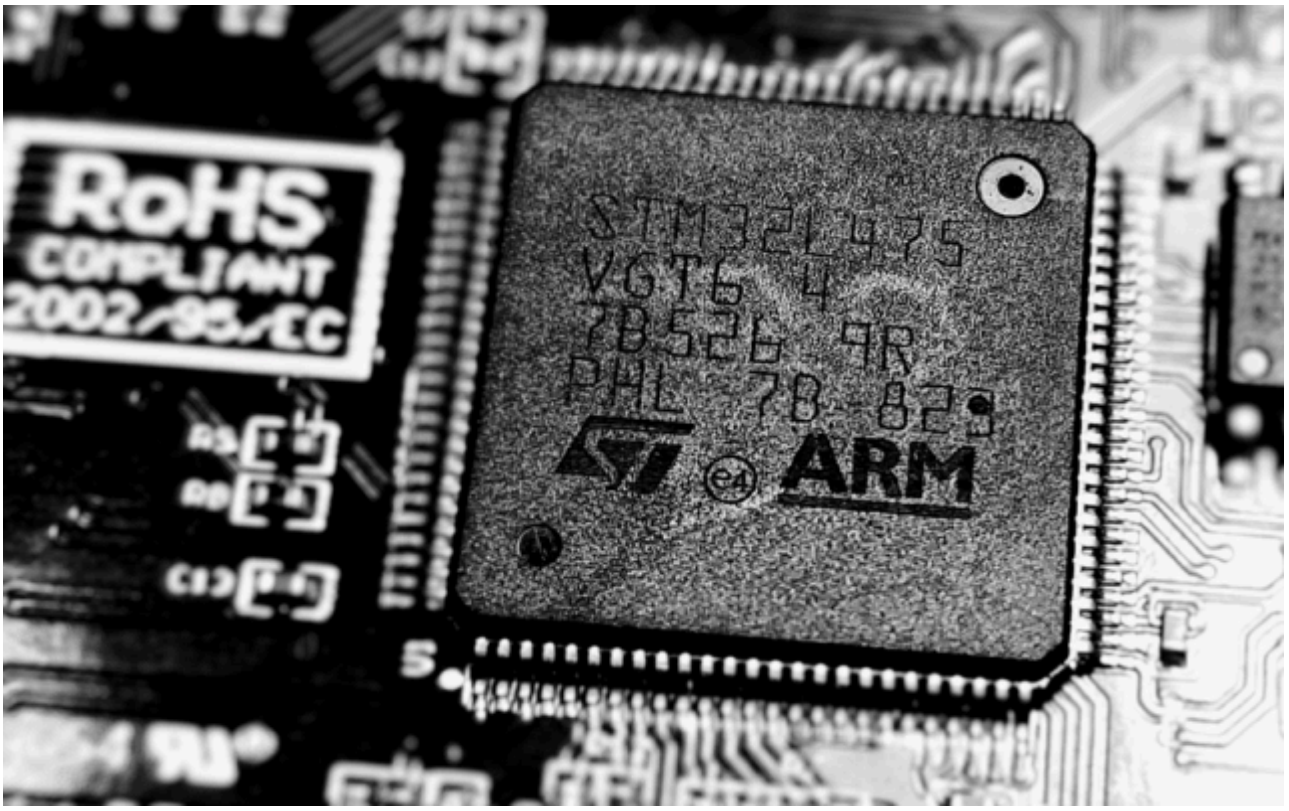
ASSOCIAZIONE NO PROFIT

The Heart of Embedded Systems

Getting Started

Theory, Code and Applications with STM32 Microcontrollers

Mauro D'Angelo



Index

Perlatecnica is a non-profit association for social promotion founded in 2014 by a group of engineering and computer science enthusiasts, with the aim of spreading digital culture through training, research and dissemination activities. Its volunteers – engineers, computer scientists and simple enthusiasts – organize seminars, workshops and online courses, creating freely accessible teaching materials and e-learning platforms.

With this book we want to carry on the same mission: to make the knowledge about STM32 microcontrollers and the basic technologies of electronics and physical computing accessible to everyone, from students to professionals. Through practical examples and step-by-step guides, the intent is to transform theoretical concepts into operational skills, stimulating the creativity and autonomy of those who approach these topics for the first time.

https://www.perlatecnica.it/

Introduction

This book is designed to take you from theory to your first practical experiences, with exercises and code snippets ready to use. The goal is that, at the end of the reading, you will not only understand the operating principles of STM32 microcontrollers, but also be able to apply them independently to your projects. A book can never be exhaustive of the topic covered, but a book like this can give the reader the appropriate ideas to conduct personal research and insights.

The code for the examples is available at the following link:

https://github.com/Perlatecnica/TheHeartOfEmbeddedSystems/tree/master/GettingStarted/WS_CoreOfEmbeddedSystems

In the following chapters you will find:

- **Chapter 1: Microcontrollers**
  Introduction to microcontrollers, CPU bit count, addressing and memory map, alternate functions and their importance, sensors and actuators, digital and analog signals, A/D conversion and communication protocols (SPI, I2C, Serial).

- **Chapter 2: The Nucleo F401RE board**
  STM32 naming convention, Nucleo-F401RE hardware features, pin labels, installation and configuration of STM32CubeIDEand introduction to CubeMX.

- **Capitolo 3: GPIO output**
  Theory of how a GPIO works, project setup, compilation and programming, LED flashing management.

- **Chapter 4: Serial Communication**
  Theory of how serial communication works, project configuration, compilation and programming, LED management through a serial command.

- **Capitolo 5: GPIO input – polling**
  Reading a configured GPIO in input, handling the user button in polling, learning about pull-up/pull-down resistors.

- **Capitolo 6: GPIO input – interrupt**
  Generating and managing interrupts from digital pins, configuring and prioritizing interrupts.

- **Chapter 7: Analog input – ADC**
  Analog to digital conversion theory, ADC configuration on STM32F401RE and main applications.

- **Chapter 8: Timers**
  Timer basics, configuration and practical use to generate timed events.

- **Chapter 9: PWM**
  Pulse width modulation fundamentals, PWM configuration on STM32 and application examples.

Chapter 1 Microcontrollers

Introduction to Microcontrollers

A **microcontroller** is an integrated electronic device that combines a processor (CPU), memory, and input/output (I/O) peripherals on a single chip. Microcontrollers are designed for embedded applications, which are electronic systems designed to perform specific tasks.

A microcontroller is composed of:

- **CPU (Central Processing Unit):** executes the program instructions. It is typically based on architectures such as ARM Cortex, AVR, or RISC.
- **Memory:**
  - **Flash:**To store the program.
  - **RAM:** For temporary data.
  - **EEPROM (optional):** For permanent and editable data
- **Peripherals:**
  - **GPIO:** General purpose input/output pins for communicating with external devices.
  - **ADC/DAC:** Signal converters, from analog to digital and vice versa.
  - **Timer:** For timing and counting functions.
  - **Communication Bus - UART, I2C, SPI, CAN:** For internal communications or with external devices.
- **Oscillator:**
  - Generates the clock to synchronize the operations of the microcontroller.

The characteristics on which microcontrollers are typically compared are:

- **Resolution:** That is the n.the number of bits of the CPU (eg 8-bit, 16-bit, 32-bit).
- **Clock frequency:** Operating speed of the microcontroller, measured in MHz or GHz.
- **Energy consumption:** Essential for battery or IoT applications.

The number of CPU bits

The **CPU bit count** of a microcontroller (for example, 8-bit, 16-bit, 32-bit) indicates the size of the data that it can process in a single instruction cycle, as well as the width of the internal registers and, generally, the architecture of the computing unit. It is a fundamental parameter that affects the performance, capabilities and efficiency of the microcontroller.

This number represents:

- **Size of operations:**
  - A CPU at **8-bit** can process numbers or instructions of up to 8 bits (1 byte) at a time.
  - A CPU at **16-bit** or **32-bit** can process larger numbers or instructions (up to 16 or 32 bits), making it more efficient for complex calculations.
- **Width of registers:**
  - The CPU's internal registers are the same width as the number of bits in the CPU.
  - For example, a CPU at 32-bit it will have 32-bit registers to store data and addresses.
- **Memory Address Management:** A CPU with a larger number of bits can handle a wider range of addresses. For example, an 8-bit CPU can address

  $$2^8 = 256$$

  memory locations, while a 32-bit CPU can address

  $$2^{32} = 4.294.967.296$$

  that is, approximately 4 billion rentals.

The following table shows a comparison between the various architectures

Characteristic	8-bit	16-bit	32-bit
Data size	Up to 8 bits	Up to 16 bits	Up to 32 bit
Memory addresses	256 locations	65,536 locations	4 billion rentals
Efficiency in calculations	Limited	Improve	Excellent
Examples of use	Simple systems	Basic Automation	Advanced Applications

For example, an 8-bit microcontroller, such as the ATmega328 (used in Arduino Uno), is ideal for simple tasks like turning on an LED or reading a sensor. However, a 32-bit microcontroller, such as the**STM32F401RE**, is best suited for complex applications, such as motor control or real-time signal processing.

In short, the number of CPU bits is a crucial parameter that determines the computing power, speed, and applications for which the microcontroller is best suited.

What does a microcontroller address?

When talking about addressing in a microcontroller, it refers to the CPU's ability to access specific memory locations. The number of bits used for addressing determines the amount of memory that can be addressed, or "reached" by the CPU. A microcontroller's memory can be:

- **Program Memory (Flash):** Contains the executable program code. The CPU needs to access the instructions to execute the code, so it uses the addressing system to find the location of each instruction.
- **Data Memory (RAM):** Used to store temporary data, variables, and stacks. Addressing allows the CPU to read and write data to specific RAM locations.
- **Peripheral Memory (I/O):** It includes peripheral registers like UART, ADC, GPIO, etc. These registers are mapped to a portion of the memory address, and the CPU manages them as part of the addressing scheme.
- **EEPROM (optional):** For storing permanent data or configurations. This memory can also be addressed by the CPU.

A microcontroller like the**STM32F401RE**It has a 32-bit CPU, which means that its address bus is 32 bits ‘wide’.

This allows it to generate addresses between **0x00000000** and **0xFFFFFFFF**, that is 2^{32} memory locations, for a total of **4 GB** addressable. However, the available physical memory is much less than this theoretical limit.

- **Flash Memory (Program Memory):** 512 KB, da 0x08000000 a 0x0807FFFF.
- **RAM (Data Memory):** 96 KB, da 0x20000000 a 0x20017FFF.
- **Peripherals:** Mapped to a specific range (e.g. GPIO, UART).
- **System Memory:** A reserved area for the bootloader and other system functions.

How Addressing Works

The CPU accesses these memories using addresses. Each address identifies a specific location or register. Let's look at some examples step by step.

Suppose the microcontroller needs to read a variable stored in RAM. The variable is located at a specific address, for example 0x20000004.

- **Address generation:**The CPU calculates or receives the address 0x20000004 and sends it to address bus.
- **Memory access:** The control unit checks that the address is within the RAM range (from 0x20000000 a 0x20017FFF). If the address is valid, it accesses the corresponding memory location.
- **Data recovery:** The CPU reads the data present in0x20000004and stores it in an internal log for further processing.

Now suppose that the microcontroller needs to write an instruction into the Flash Memory, at the address 0x08000010:

- **Generated address:** The CPU sends the address 0x08000010to the address bus.
- **Flash Access:**The system checks that the address is in the Flash range (from 0x08000000 a 0x0807FFFF). If yes, it allows writing (provided the Flash is in writable mode).
- **Data writing:**The instruction is written to the specified location.

Suppose instead that I want to control a peripheral, for example an LED controlled by a GPIO pin, and the GPIO register is at address 0x40020014.

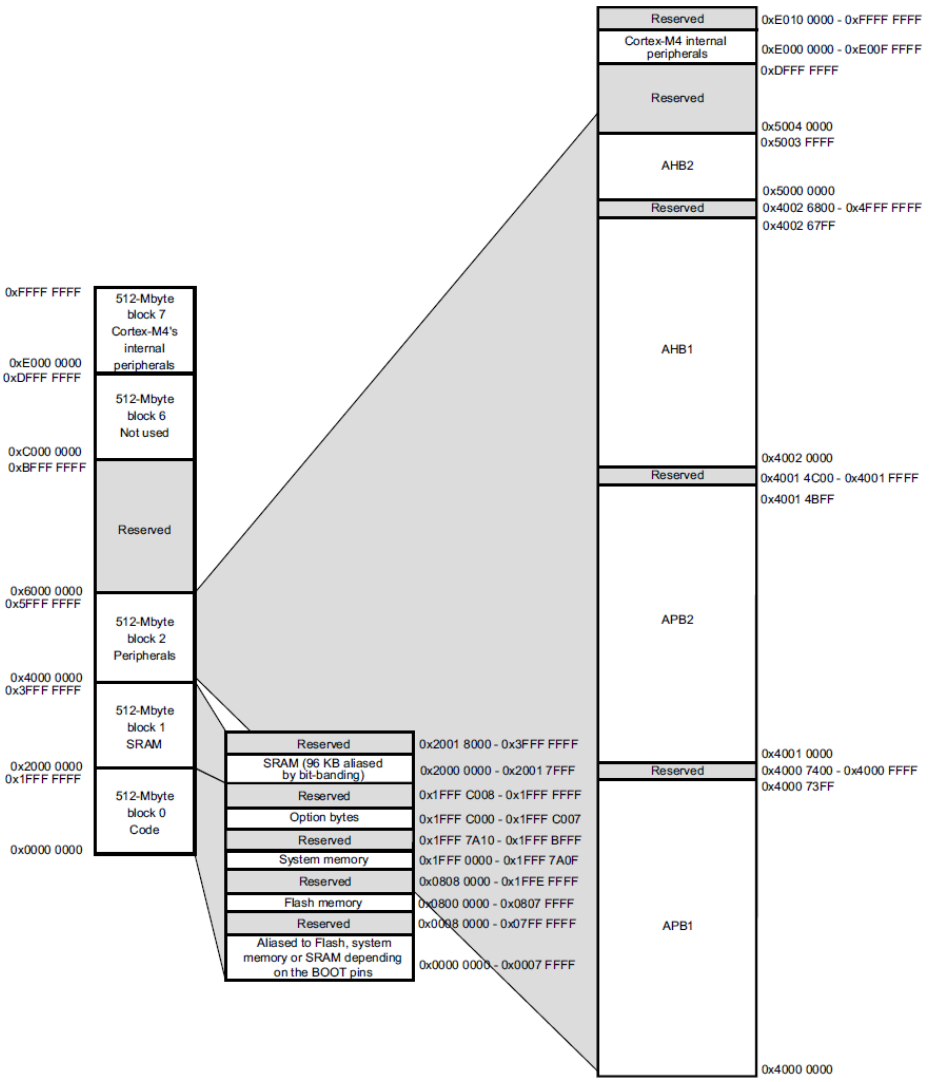
- **GPIO register address:** The CPU generates the address 0x40020014, which corresponds to a GPIO register for setting the state of a pin.
- **Writing to the register:** The CPU writes a value (for example,1 to turn on the LED) in the register.

The memory map

In the STM32F401RE, different memory areas are mapped to a specific range:

Address Range	Memory Type	Description
0x08000000 - 0x0807FFFF	Flash Memory	Program Memory (512 KB)
0x20000000 - 0x20017FFF	RAM	Data Memory (96 KB)
0x40000000 - 0x500607FF	Peripherals	Peripheral registers (GPIO, UART, etc.)
0x1FFF0000 - 0x1FFF77FF	System Memory	Bootloader and other system functions

The following figure, extracted from the datasheet of the STM32F401RE microcontroller, explores in detail the memory map of the device.



In practice, addressing allows the CPU to manage all components (RAM, Flash, peripherals) using a single address system. Each operation is driven by the microcontroller's logic and mapped to a portion of the address bus, making access to memory and devices simple and uniform.

Applications

The most common applications of Microcontrollers are:

- **Home automation:** Control lights, thermostats, smart locks.
- **Automotive:** Engine control systems, airbags, ABS.
- **Industry:** Automation of processes and machinery.
- **IoT (Internet of Things):** Connected devices for data collection and analysis.
- **Consumer Electronics:** Home appliances, interactive toys, smart watches.

Microcontrollers are the basis of modern embedded electronics, with applications ranging from home automation to industry. Understanding their structure and programming is essential to developing innovative technological solutions.

Alternate Functions in Microcontrollers

Modern microcontrollers are extremely versatile devices, featuring numerous pins that can be configured to perform different functions. These multiple configurations, often called *alternate functions*,allow the microcontroller to adapt to a wide range of applications, optimizing the use of available hardware resources.

What are theAlternate Functions

A microcontroller pin is not limited to just one function. In addition to the basic function (typically a digital input or output), many pins can be used for other functions, such as:

- **Analog Inputs/Outputs** (ADC o DAC)
- **Serial communications** (UART, SPI, I2C, CAN, ecc.)
- **PWM (Pulse Width Modulation)**to control motors or LEDs
- **Timer inputs**
- **Special functions of the microcontroller**, come clock o debug

These alternative functions are configurable through internal control registers of the microcontroller.

Why they are important

The alternate functions allow you to:

- **Optimize your use of pins:** A microcontroller with a limited number of pins can offer a lot of functionality thanks to dynamic configuration.
- **Reduce system size:** By implementing more functionality on fewer pins, PCB size and manufacturing costs are reduced.
- **Provide design flexibility:** The developer can choose how to configure the pins to best suit the application.

How They Work

The alternate functions are managed via:

- **GPIO Configuration Registers:** Each pin has one or more configuration registers associated with it. These registers specify:
    - Pin mode (digital, analog, etc.)
    - Pin speed
    - Pull-up/pull-down resistors
    - Alternative function selected
- **Mapping of Alternate Functions:** Each pin has a mapping table that specifies which alternate functions are available. For example, the microcontroller datasheet provides a matrix that lists the alternate functions of each pin.

- **Internal Multiplexer:** Inside the microcontroller, the pins are connected to a hardware multiplexer. The configuration of the registers selects which function is enabled for that pin.

The alternate functions are a fundamental aspect to make the most of a microcontroller. A correct configuration is essential to ensure the optimal functioning of the application. The developer must always consult the datasheet and use software tools (such as HAL or specific drivers) to configure these features efficiently.

Alternate functions of the STM32F401RE microcontroller

For the convenience of the reader, the table of all the alternate functions offered by each pin of the STM32F401RE microcontroller. The tables are extracted from the datasheet which we recommend consulting.

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port A	PA0	-	TIM2_CH1/ TIM2_ETR	TIM5_CH1	-	-	-	-	USART2_ CTS	-	-	-	-	-	-	-	EVENT OUT
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-	USART2_ RTS	-	-	-	-	-	-	-	EVENT OUT
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-	USART2_ TX	-	-	-	-	-	-	-	EVENT OUT
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-		-	USART2_ RX	-	-	-	-	-	-	-	EVENT OUT
	PA4	-	-	-	-	-	SPI1_NSS	SPI3_NSS/ I2S3_WS	USART2_ CK	-	-	-	-	-	-	-	EVENT OUT
	PA5	-	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_SCK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA6	-	TIM1_BKIN	TIM3_CH1	-	-	SPI1_ MISO	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA7	-	TIM1_CH1N	TIM3_CH2	-	-	SPI1_ MOSI	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA8	MCO_1	TIM1_CH1	-	-	I2C3_SCL	-	-	USART1_ CK	-	-	OTG_FS_ SOF	-	-	-	-	EVENT OUT
	PA9	-	TIM1_CH2	-	-	I2C3_ SMBA	-	-	USART1_ TX	-	-	OTG_FS_ VBUS	-	-	--	-	EVENT OUT
	PA10	-	TIM1_CH3	-	-	-	-	-	USART1_ RX	-	-	OTG_FS_I D	-	-	-	-	EVENT OUT
	PA11	-	TIM1_CH4	-	-	-	-	-	USART1_ CTS	USART6_ TX	-	OTG_FS_ DM	-	-	-	-	EVENT OUT
	PA12	-	TIM1_ETR	-	-	-	-	-	USART1_ RTS	USART6_ RX	-	OTG_FS_ DP	-	-	-	-	EVENT OUT
	PA13	JTMS_ SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA14	JTCK_ SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA15	JTDI	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_NSS	SPI3_NSS/ I2S3_WS	-	-	-	-	-	-	-	-	EVENT OUT

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port B	PB0	-	TIM1_CH2N	TIM3_CH3	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PB1	-	TIM1_CH3N	TIM3_CH4	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PB2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PB3	JTDO-SWO	TIM2_CH2	-	-	-	SPI1_SCK	SPI3_SCK/ I2S3_CK	-	-	I2C2_SDA	-	-	-	-	-	EVENT OUT
	PB4	JTRST	-	TIM3_CH1	-	-	SPI1_ MISO	SPI3_MISO	I2S3ext_S D	-	I2C3_SDA	-	-	-	-	-	EVENT OUT
	PB5	-	-	TIM3_CH2	-	I2C1_ SMBA	SPI1 _MOSI	SPI3_MOSI/ I2S3_SD	-	-	-	-	-	-	-	-	EVENT OUT
	PB6	-	-	TIM4_CH1	-	I2C1_SCL	-	-	USART1_ TX	-	-	-	-	-	-	-	EVENT OUT
	PB7	-	-	TIM4_CH2	-	I2C1_SDA	-	-	USART1_ RX	-	-	-	-	-	-	-	EVENT OUT
	PB8	-	-	TIM4_CH3	TIM10_CH1	I2C1_SCL	-	-	-	-	-	-	-	SDIO_ D4	-	-	EVENT OUT
	PB9	-	-	TIM4_CH4	TIM11_CH1	I2C1_SDA	SPI2_NSS/I 2S2_WS	-	-	-	-	-	-	SDIO_ D5	-	-	EVENT OUT
	PB10	-	TIM2_CH3	-	-	I2C2_SCL	SPI2_SCK/I 2S2_CK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PB12	-	TIM1_BKIN	-	-	I2C2_ SMBA	SPI2_NSS/I 2S2_WS	-	-	-	-	-	-	-	-	-	EVENT OUT
	PB13	-	TIM1_CH1N	-	-	-	SPI2_SCK/I 2S2_CK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PB14	-	TIM1_CH2N	-	-	-	SPI2_MISO	I2S2ext_SD	-	-	-	-	-	-	-	-	EVENT OUT
	PB15	RTC_ REFN	TIM1_CH3N	-	-	-	SPI2_MOSI /I2S2_SD	-	-	-	-	-	-	-	-	-	EVENT OUT

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port C	PC0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC2	-	-	-	-	-	SPI2_ MISO	I2S2ext_SD	-	-	-	-	-	-	-	-	EVENT OUT
	PC3	-	-	-	-	-	SPI2_MOSI /I2S2_SD	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC6	-	--	TIM3_CH1	-	-	I2S2_MCK	-	-	USART6_ TX	-	-	-	SDIO_ D6	-	-	EVENT OUT
	PC7	-		TIM3_CH2	-	-	-	I2S3_MCK	-	USART6_ RX	-	-	-	SDIO_ D7	-	-	EVENT OUT
	PC8	-	-	TIM3_CH3	-	-	-	-	-	USART6_ CK	-	-	-	SDIO_ D0	-	-	EVENT OUT
	PC9	MCO_2	-	TIM3_CH4	-	I2C3_SDA	I2S_CKIN	-	-	-	-	-	-	SDIO_ D1	-	-	EVENT OUT
	PC10	-	-	-	-	-	-	SPI3_SCK/ I2S3_CK	-	-	-	-	-	SDIO_ D2	-	-	EVENT OUT
	PC11	-	-	-	-	-	I2S3ext_ SD	SPI3_MISO	-	-	-	-	-	SDIO_ D3	-	-	EVENT OUT
	PC12	-	-	-	-	-	-	SPI3_MOSI/ I2S3_SD	-	-	-	-	-	SDIO_ CK	-	-	EVENT OUT
	PC13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PC15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port D	PD0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD2	-	-	TIM3_ETR	-	-	-	-	-	-	-	-	-	SDIO_ CMD	-	-	EVENT OUT
	PD3	-	-	-	-	-	SPI2_SCK/ I2S2_CK	-	USART2_ CTS	--	-	-	-	-	-	-	EVENT OUT
	PD4	-	-	-	-	-	-	-	USART2_ RTS		-	-	-	-	-	-	EVENT OUT
	PD5	-	-	-	-	-	-	-	USART2_ TX	-	-	-	-	-	-	-	EVENT OUT
	PD6	-	-	-	-	-	SPI3_MOSI /I2S3_SD	-	USART2_ RX	-	-	-	-	-	-	-	EVENT OUT
	PD7	-	-	-	-	-	-	-	USART2_ CK	-	-	-	-	-	-	-	EVENT OUT
	PD8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD12	-	-	TIM4_CH1	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD13	-	-	TIM4_CH2	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD14	-	-	TIM4_CH3	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PD15	-	-	TIM4_CH4	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port E	PE0	-	-	TIM4_ETR	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE1	-	TIM1_CH2N	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE2	TRACECL K	-	-	-	-	SPI4_SCK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE3	TRACED0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE4	TRACED1	-	-	-	-	SPI4_NSS	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE5	TRACED2	-	-	TIM9_CH1	-	SPI4_MISO	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE6	TRACED3	-	-	TIM9_CH2	-	SPI4_MOSI	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE7	-	TIM1_ETR	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE8	-	TIM1_CH1N	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE9	-	TIM1_CH1	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE10	-	TIM1_CH2N	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE11	-	TIM1_CH2	-	-	-	SPI4_NSS	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE12	-	TIM1_CH3N	-	-	-	SPI4_SCK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE13	-	TIM1_CH3	-	-	-	SPI4_MISO	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE14	-	TIM1_CH4	-	-	-	SPI4_MOSI	-	-	-	-	-	-	-	-	-	EVENT OUT
	PE15	-	TIM1_BKIN	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port H	PH0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PH1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT

Physical Computing

Physical *Computing* is the intersection between the physical and digital worlds, where electronic devices, such as microcontrollers, are used to interact with the environment through sensors, actuators and input/output devices. This approach is essential for developing interactive applications, innovative prototypes and IoT solutions.

Microcontrollers are essential components of physical computing due to their ability to:

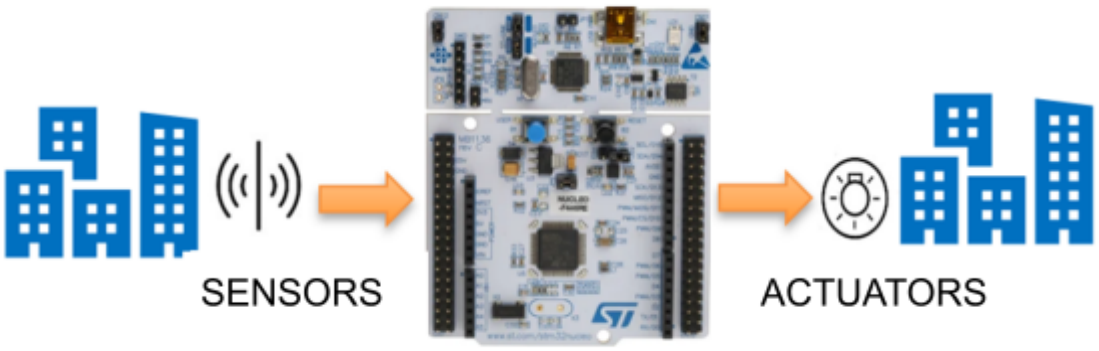
- **Detects the environment** through sensors (e.g. temperature, light, pressure).
- **Process data** coming from the sensors.
- **Interact with the environment** via actuators (e.g. motors, LEDs, speakers).
- **Communication** with other devices via standard protocols (e.g. I2C, SPI, UART, Wi-Fi).

A physical computing system is based on three main components:

- **Input:**
  - Sensors to detect physical or environmental data (e.g. temperature, humidity, light).
  - Input devices such as buttons, potentiometers, joysticks, or touch sensors.
- **Processing:**
  - Microcontroller that processes data and decides what actions to take.
  - Algorithms programmed to analyze and respond to inputs received.
- **Output:**
  - Actuators to respond to inputs (e.g. motors, solenoid valves, relays).
  - Visual or acoustic feedback (e.g. LED, LCD display, buzzer).

Examples of Physical Computing Projects are:

- **Weather StationIoT:**
  - **Input:**Temperature, humidity and pressure sensors.
  - **Processing:**Calculate environmental data and send to the cloud.
  - **Output:**View on LCD display or online dashboard.
- **Robot Mobile:**
  - **Input:**Distance sensors (e.g. ultrasonic) and control buttons.
  - **Processing:**Obstacle avoidance algorithms.
  - **Output:**Control of motors for movement.
- **Smart Home:**
  - **Input:**PIR sensors to detect motion and smart switches.
  - **Processing:**Automatic on/off logic.
  - **Output:**Control lights, shutters or connected devices.

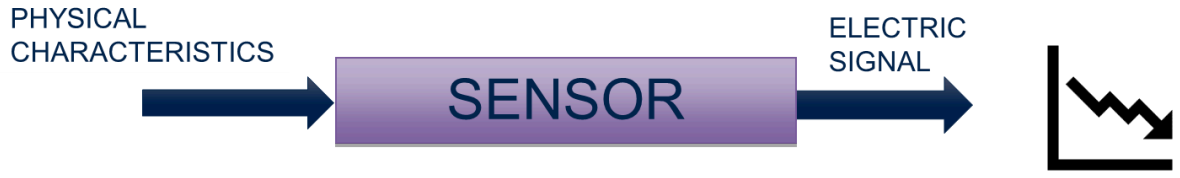


So the sensors perceive the environment and provide this information to the microcontroller, which based on the program being run, can perform actions via the actuators.

Physical computing with microcontrollers offers endless possibilities to create innovative solutions that interact with the physical world. Understanding the basic principles and experimenting with practical projects is the best way to develop skills in this field.

Sensors

A sensor is a device that detects physical or chemical changes in the environment and converts them into electrical signals that can be read by an electronic system, such as a microcontroller. Sensors are used in a wide range of applications, from industrial automation to the Internet of Things (IoT), to gather information about the surrounding environment.

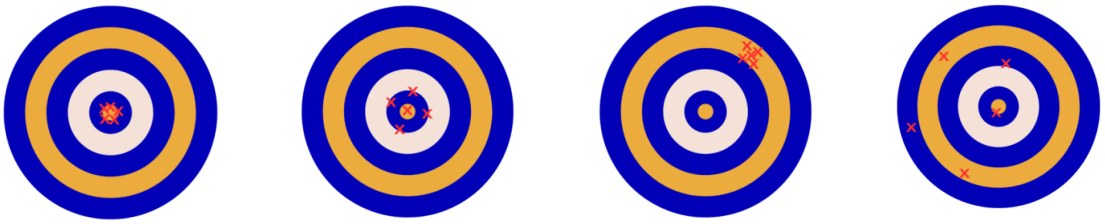


A sensor has the following main characteristics:

- **Measured Size**: It is the physical or chemical variable that the sensor detects, such as temperature, humidity, pressure, light, position, gas, or force.
- **Exit**The detected quantity is converted into an electrical signal, which can be:
  - **Analog**: A continuous signal (for example, a voltage value proportional to the measured quantity).
  - **Digital**: A discrete signal, often transmitted using protocols such as I2C, SPI, or UART.
- **Range**: The range of values that the sensor can measure.
- **Accuracy and Resolution**:
  - **Accuracy**: The sensor's ability to provide a value close to the real one.

    - **Precision**: the ability to repeatedly indicate the same measurement value, regardless of whether or not it corresponds to the true value.
    - **Resolution**: The level of detail with which the sensor detects changes.
- **Sensitivity**: The ability of the sensor to respond to small changes in the measured quantity.

The target example helps to easily understand the meanings of accuracy and precision of a sensor



High accuracy High precision | High accuracy Low precision | Low accuracy High precision | Low accuracy Low precision

Example: Temperature Sensor on IKS4A1 Expansion Board

The **X-NUCLEO-IKS4A1** is an expansion board designed for the STM32 Nucleo platforms, featuring advanced sensors for motion detection and environmental monitoring. Among the integrated sensors there is theSTTS22H,a high-precision digital sensor for temperature measurement.

The technical characteristics of theSTTS22H:

- **Temperature Measurement Range**: and -40 °C at +125 °C.
- **Accuracy**: ±0.5°C within the standard operating range.
- **Resolution**: 16 bits (up to 0.0625°C per increment).
- **Communication Interface**:
    - **I²C**with configurable slave address.
    - Supports speeds up to 1 MHz (I²C Fast Mode Plus).
- **Energy Consumption**:
    - **1,75 µA** in active mode (at 1 Hz).
    - **0,5 µA** in standby mode.

The sensor STTS22H measures temperature using a diode junction system. Temperature variations affect the electrical behavior of the diode, and the sensor processes these variations to provide highly accurate digital values.

The sensor STTS22H easily integrates with any STM32 Nucleo board thanks to the layout compatibility Arduino Uno R3 present on the X-NUCLEO-IKS4A1. Communication occurs via I²C bus.

So in summary, the sensor perceives the temperature through a variation in the electrical behavior of the diode and makes the temperature data available through an I²C protocol.

The actuators

The **Actuators** are devices that transform a control signal, usually electrical, into a physical action. They are used to perform movements, apply forces or generate specific effects such as light, heat or sound. Actuators represent the "active" part of an automation or control system, interacting directly with the surrounding environment.

Actuators can be classified according to:

- **Energy used:**
  - **Electric:** Motors, electromagnets, solenoids.
  - **Plumbers:** Pistons and cylinders powered by pressurized fluids.
  - **Tires:** Actuators powered by compressed air.
- **Movement type:**
  - **Linear:** They produce a rectilinear movement (e.g. pistons).
  - **Rotating:**They produce a rotary movement (e.g. DC motors, steppers).
- **Function performed:**
  - **Positioning:** To adjust the position of an object.
  - **By force:** To apply a specific force.

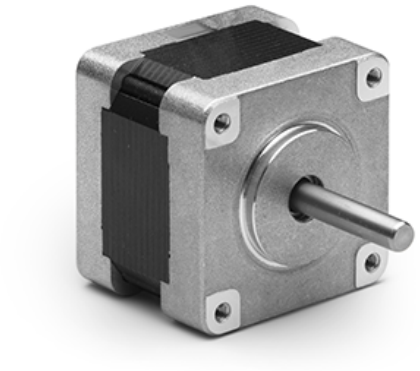
Electric Actuators

Electric actuators use electrical energy to generate motion or physical effects. Examples of this type of actuators are:

- **DC (Direct Current) Motors:**

    - Continuous rotary motion.
    - Simple to control via PWM signals.
    - Used in robotics, vehicles, household appliances.

- **Stepper Motors:**



    - Rotary movement with discrete steps.
    - Ideal for applications requiring precise position control, such as 3D printers or CNC machines.

- **Actuators:**
    - They provide precise control of position, speed and torque.
    - They combine a DC motor, a gearbox and a control system.
    - Used in arms robotics and automation.

- **Solenoidi:**

- They generate linear motion when powered.
- Used in electronic locks, valves and locking mechanisms.

Hydraulic Actuators

- **Characteristics:**
  - High strength and robustness.
  - Ideal for heavy industrial applications (e.g. construction machinery).
- **Examples:**
  - Hydraulic cylinders for lifting and compression.
  - Hydraulic steering systems in vehicles.

Pneumatic Actuators

Pneumatic actuators use compressed air to generate movement.

- **Characteristics:**
  - Fast, cheap and safe.
  - Suitable for industrial environments and automated systems.
- **Examples:**
  - Pneumatic pistons for assembly machinery.

    - Pneumatically controlled valves.

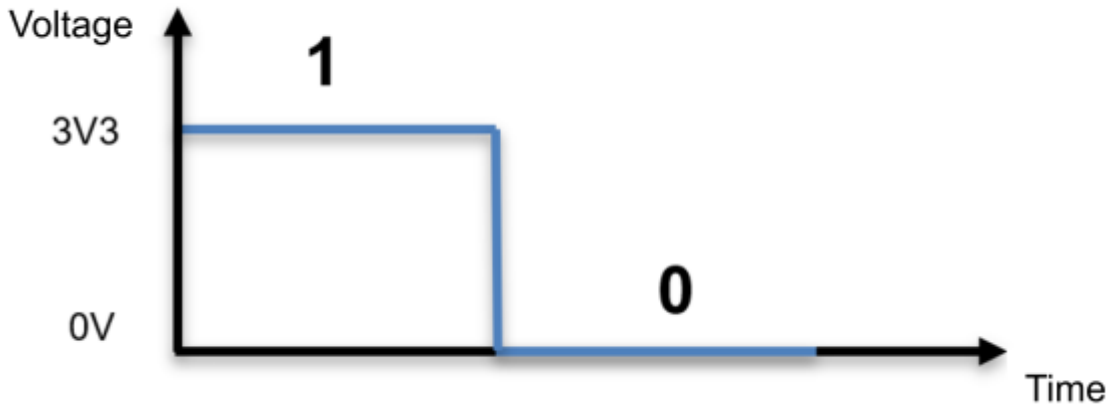
Thermal Actuators

They use temperature variations to generate movement or changes in state.

- **Examples:**
    - Wax actuators for thermostats.
    - Piezoelectric elements that deform in response to heat.

Digital signals

A digital signal is a type of signal that represents information using discrete values, typically binary (0 and 1). Unlike analog signals, which vary continuously, a digital signal only has a finite number of states, making it easier to process and transmit in electronic systems.



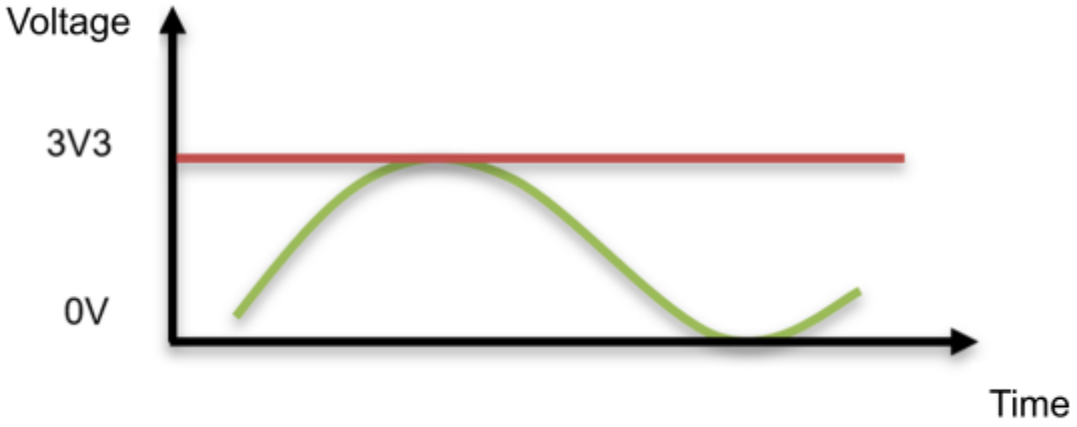
The main features are:

- **Discrete values:** Usually represented by voltage levels, for example:
  - 0: Low voltage (e.g. 0V).
  - 1: Tensione alta (es. 3.3V o 5V).
- **Noise immunity:** Digital signals are less sensitive to noise than analog signals, since the system detects only two distinct states.
- **Ease of processing:** They are used in digital systems, such as microcontrollers and processors, where logical and arithmetic operations can be performed precisely.

Digital signals are fundamental to communication and processing in modern electronic systems, including computers, IoT devices, and telecommunication networks.

Analog signals

An analog signal is a type of signal that varies continuously over time, representing information through an infinite range of possible values within a given interval. It is commonly used to represent physical phenomena such as sound, light, or temperature.



The main features are:

- **Continuity:** The signal value can take any value within the specified range.
- **Noise sensitivity:** Analog signals are more prone to noise and distortion than digital signals.
- **Natural representation:** They are ideal for representing physical phenomena that vary continuously.

Examples of analog signals include voice captured by a microphone, light waves detected by an optical sensor, and voltage in a variable circuit.

Analog to Digital Conversion

At this point it is clear that the microcontroller works in the ‘digital world’, but interacts with an ‘analog world'. To enable this interaction it is necessary to have devices useful for signal conversion.

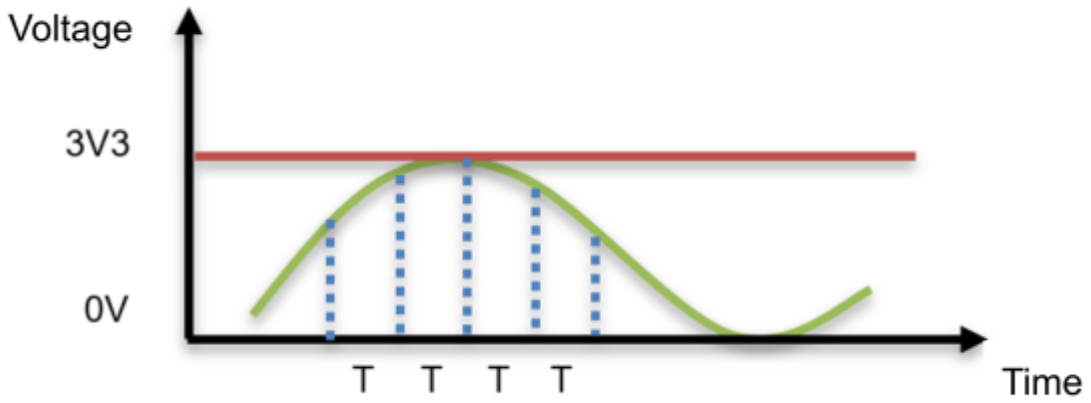
Analog-to-digital conversion (ADC) is the process of transforming a continuous analog signal into a discrete digital signal, that is, one that takes on a finite number of values that are encoded, which can be processed by digital electronic systems such as microcontrollers or computers. This process is essential for acquiring and analyzing information from the physical world, such as temperature, light, or sound.

The ADC conversion process includes three main stages:

- **Sampling:**
  - The analog signal is measured at regular time intervals, defined by the sampling rate.
  - The sampling rate must be at least twice the maximum frequency of the signal, according to the Nyquist theorem, to avoid aliasing.
- **Quantizzazione (Quantization):** The continuous values of the sampled signal are mapped to a finite number of discrete levels, determined by the resolution of the ADC. For example, a 12-bit ADC divides the analog range into:

$$2^{12} = 4096 \text{levels}$$

- **Codifica (Encoding):** Each quantized value is represented as a binary number, readable by the digital system.



The main characteristics of an ADC are:

- **Resolution:** Indicates the number of bits used to represent digital values. For example, a 12-bit ADC has a resolution of 1/4096 of the total range.

$$Step = \frac{Vref}{2^{12}} = \frac{Vref}{4096}$$

- **Voltage Range:** The voltage range the ADC can measure (e.g. 0-3.3V or ±5V).
- **Accuracy:** The difference between the actual value of the analog signal and the converted digital value.
- **Sampling Rate:** The speed at which the ADC can measure the signal, expressed in samples per second (SPS).
- **Quantization Error:** The difference introduced during the conversion between the continuous analog signal and the discrete digital signal.

Practical example: ADC on STM32F401RE

The microcontroller The STM32F401RE features a 12-bit ADC, which can convert an analog signal (for example a voltage between 0 and 3.3V) into a digital value between 0 and 4095.

The sampling rate is configurable and depends on the ADC clock frequency.

If the analog input voltage is 1.65V and the maximum reference voltage is 3.3V, the corresponding digital value is calculated as follows:

$$Digital\ value\ =\ \frac{Input\ Voltage}{Reference\ Voltage}\ *\ (2^{12}\ -\ 1)$$

$$Digital\ value\ =\ \frac{1.65}{3.3}\ *\ (4096\ -\ 1) \approx 2048$$

The digital value **2048** represents **1,65V** in the system.

The main applications of ADC

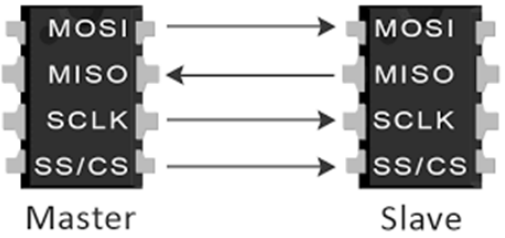
- **Audio:** Convert analog sound signals into digital data for recording or processing.
- **Sensors:** Measure physical quantities such as temperature, light, pressure, or other variables.
- **Imaging:** Convert signals from optical sensors into digital images.
- **Automation:** Control systems based on analog feedback, such as motors or HVAC systems (Heating, Ventilation & Air Conditioning).

The protocols of communication

Communication with other devices connected to the microcontroller occurs through dedicated protocols that define the connection methods and communication rules. Below we examine the main protocols,

SPI - Serial Peripheral Interface

The Serial Peripheral Interface (SPI) protocol is a high-speed synchronous serial communication method used to exchange data between microcontrollers and peripheral devices such as sensors, displays, or memories. It works in a master-slave configuration, with one master and one or more slaves.



The main features are:

- **Synchronous communication**: A clock (SCLK) synchronizes the data transfer between master and slave.
- **Full-duplex**: Data is transmitted and received simultaneously on separate lines.
- **High speed**: It supports faster communication than other serial protocols such as I²C or UART.
- **Simple topology**: Relatively easy to implement, but requires more cabling than other protocols.

The main SPI lines are:

- **SCLK (Serial Clock)**: Generated by the master, synchronizes data transfer.
- **MOSI (Master Out Slave In)**: Line on which the master sends data to the slave.
- **MISO (Master In Slave Out)**: Line on which the slave sends data to the master.
- **SS/CS (Slave Select/Chip Select)**: Dedicated line to select which slave is active.

It works according to the following steps:

- **Initial setup**: The master sets the main parameters, such as clock speed, clock polarity (CPOL), clock phase (CPHA) and line roles. Each slave is connected to the master via dedicated lines (MOSI, MISO, SCLK), but each slave has its own SS/CS line.

- **Slave selection**: The master brings the SS/CS line of the target slave to a low level (in the case of active low logic). The other slaves ignore the communication by keeping their own SS/CS line high.
- **Data exchange**: The master generates clock signals (SCLK) and transfers data via MOSI. The slave responds simultaneously by sending data via MISO. Data transfer occurs one bit at a time, synchronized with the SCK signal.
- **End of communication**: The master brings the SS/CS line to a high level, signaling the end of communication with that slave.

Advantages of SPI protocol:

- **High speed**: Transfer speeds that can reach tens of MHz.
- **Full-duplex**: Allows simultaneous bidirectional data transfer.
- **Simple hardware**: Easy to implement with dedicated lines.

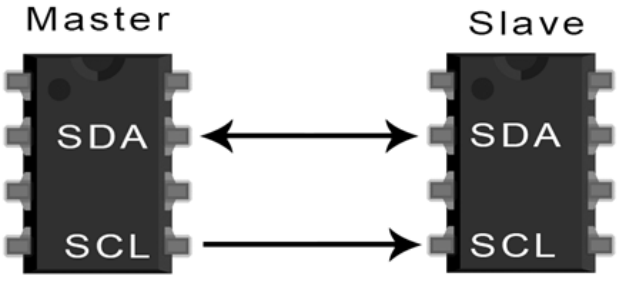
The disadvantages of the protocol are:

- **High pin usage**: Each slave requires a dedicated SS/CS line, which can become problematic with many devices.
- **No built-in addressing or error checking**: Requires additional management via software.

SPI is ideal for applications that require fast communication and a limited number of peripherals, such as sensors or high-speed memory devices.

I2C

The I²C (Inter-Integrated Circuit) protocol is a multi-master, multi-slave, synchronous serial communication protocol designed for low-speed communication between integrated circuits. It is widely used to connect microcontrollers to sensors, displays, and other peripheral devices.



The main features are:

- **Synchronous communication**: Data transfer is synchronized via a clock line (SCL).
- **Two-wire communication**: Requires only two lines, reducing pin usage compared to other protocols.
- **Address-based communication**: Each slave has a unique address, allowing communication with many devices using the same lines.

The main I²C lines are:

- **SCL (Serial Clock Line)**: The clock signal generated by the master to synchronize data transfer.
- **SDA (Serial Data Line)**: A two-way line used to send and receive data.
    - Both lines are open-drain and require pull-up resistors to maintain a default high state.

The operation is defined by the following steps:

- **Initial setup**: The master configures the clock speed and defines communication parameters, such as addressing mode (7-bit or 10-bit).
- **Start condition**: The master generates a start condition by pulling the SDA line low while SCL remains high, signaling the start of a communication frame.
- **Addressing**:
    - The master sends the address of the target slave, followed by a read/write bit indicating the type of operation.
    - All devices on the bus listen, but only the addressed slave responds by lowering SDA (acknowledge or ACK).
- **Data Transfer**:
    - Master and slave exchange data in 8-bit (byte) packets, each followed by an acknowledgment (ACK) or a no-acknowledge (NACK).
    - Clock pulses (SCL) control when data is read or written.
- **Stop condition**: The master generates a stop condition by releasing SDA high while SCL remains high, signaling the end of communication.

The I²C protocol offers the following advantages:

- **Low pin usage**: Only two lines are required for communication, regardless of the number of devices.
- **Multi-master capability**: Multiple masters can coexist on the same bus.
- **Addressing**: Devices can share the same bus thanks to unique addresses, without additional lines.

The disadvantages are instead:

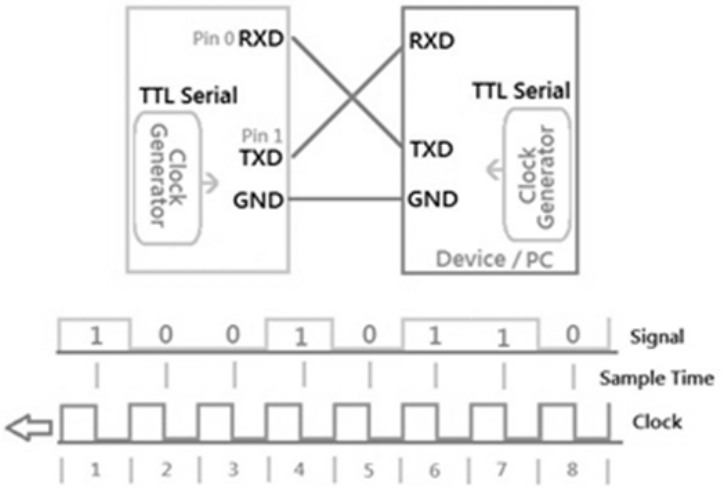
- **Slower than SPI**: Speeds limited to a few MHz, depending on the implementation.

- **Limited bus length**: Due to capacity issues, the bus is only suitable for short distances.
- **Complexity**: Requires acknowledgment and addressing management, increasing software load.

I²C is ideal for applications that require low to moderate transfer rates and connecting multiple devices using a minimum number of wires, such as in embedded systems and IoT applications. Its simplicity and scalability make it a popular choice for communicating between components on a PCB.

Serial

Serial protocol (UART - Universal Asynchronous Receiver-Transmitter) is a simple and widely used communication protocol that allows data transfer between devices, such as microcontrollers, sensors, or computers, using asynchronous serial communication. It works without a clock signal and relies on a predefined configuration for synchronization.



The main features are:

- **Asynchronous communication**: Does not require a clock signal from a master to a slave; synchronization occurs via start and stop bits. Both devices must know the communication speed.
- **Full-duplex**: Data can be sent and received simultaneously on separate lines.
- **Point to point communication**: Typically used to directly connect two devices.
- **Configurable settings**: Communication parameters, such as baud rate, parity, data bits and stop bits, must be agreed upon by both devices.

The main serial lines are:

- **TX (Transmit)**: Line on which the device sends data.
- **RX (Receive)**: Line on which the device receives data.
- **Optional control lines**: Like CTS (Clear to Send) and RTS (Request to Send) for hardware flow control.

The serial protocol works in the following modes:

- **Configuration**: Both devices must agree on the communication parameters:
    - **Baud rate**: Communication speed in bits per second (e.g. 9600 bps).
    - **Bit in dati**: Number of bits in each data frame (e.g. 7 or 8 bits).
    - **Parity bit**: Optional bit for error checking (even, odd or no parity).
    - **Bit on stop**: Number of bits that mark the end of a data frame (e.g. 1 or 2 bits).
- **Data Framing**Each byte of data is encapsulated in a frame constructed as follows:
    - **Please start**: A single bit (low level) that signals the start of the frame.
    - **Bit in dati**: I real data (eg 8 bit).
    - **Optional parity bit**: For error checking.
    - **Bit on stop**: One or more bits (high level) that mark the end of the frame.
- **Data transmission**:
    - The transmitter sends the encapsulated data frame by frame, bit by bit.
    - The receiver samples the data line at the agreed speed (baud rate) to reconstruct the original data.
- **Error Handling**Errors such as framing or parity errors may occur, which must be handled by the receiving device.

The serial protocol offers the following messages:

- **Simplicity**: Easy to implement and requires minimal hardware.
- **Low pin usage**: Only two lines (TX and RX) are required for communication.
- **Asynchronous**: Does not require a clock line, simplifying wiring.

The disadvantages are instead:

- **Limited to two devices**: Direct connection between only two devices (unless using multiplexers).
- **No integrated addressing**: It cannot communicate natively with multiple devices.
- **Slower than synchronous protocols**: Due to the overhead of the optional start, stop and parity bits.

The serial protocol is ideal for simple, low-speed communication between two devices. Its simplicity and widespread use make it a popular choice for debugging, firmware updates, and data logging in embedded systems.

Let's recap

We explored the world of **Physical Computing**, where microcontrollers are the beating heart of systems that connect the physical and digital worlds. Through sensors, actuators and communication protocols, microcontrollers collect information from the environment, process it and respond with precise actions.

Key points to remember:

- **Connection with the environment:**
  - **Sensors**: Devices that sense physical variables (e.g. temperature, light) and convert them into electrical signals. These signals can be:
    - **Analogue**: Continuous values that require ADC conversion to be understood by the microcontroller.
    - **Digital**: Discrete values, often transmitted via standard protocols such as I²C, SPI, or UART.
  - **Actuators**: Devices that transform electrical signals into physical actions, such as movement, light or sound, closing the loop of interaction with the environment.

- **Communication protocols:**
  - **SPI**: Ideal for high-speed, full-duplex connections to peripherals such as displays and memory.
  - **I²C**: Perfect for multi-slave systems with a reduced number of wires.
  - **UART**: A simple and popular option for point-to-point communication. Each protocol has unique characteristics that make it suitable for specific scenarios, providing flexibility in design.

- **Analog to Digital Conversion (ADC):**Converting analog signals into digital signals is essential to allow the microcontroller to interact with the real world, ensuring accuracy and reliability in measurements.

The Importance of Physical Computing. The ability to integrate input, processing and output enables the creation of interactive and intelligent applications, from home automation to mobile robots, through weather stations.IoT.The secret to success lies in understanding the characteristics of the signals and choosing the most suitable communication protocol.

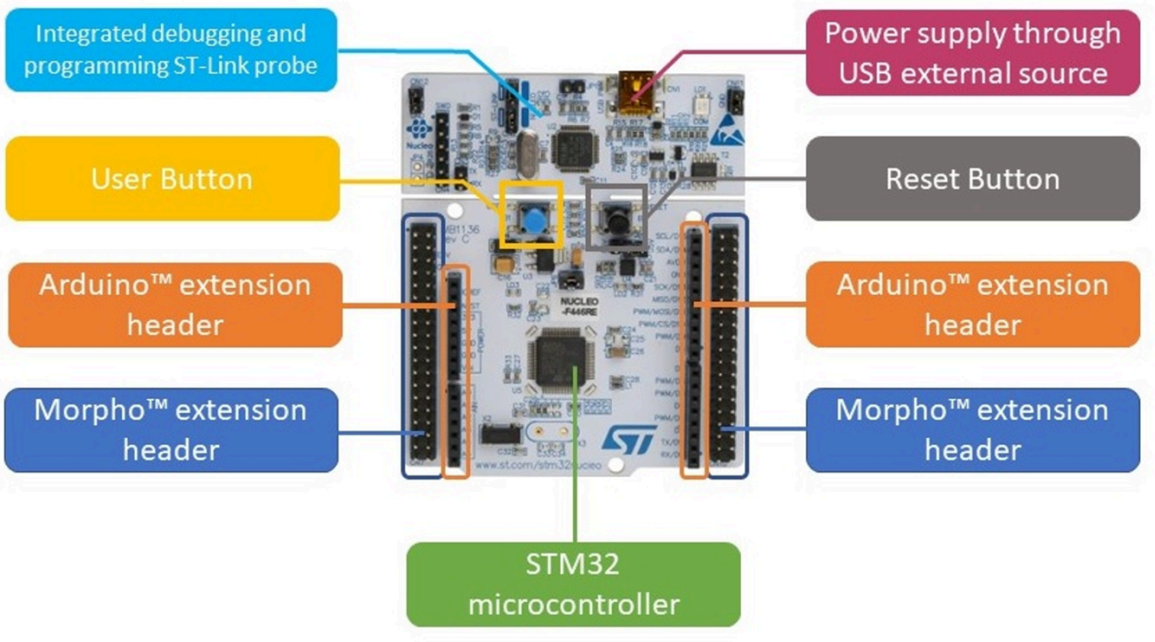
Chapter 2 The Nucleo F401RE board

The NucleoF401REis a development board designed by STMicroelectronics, based on the STM32F401RE microcontroller, part of the STM32 family. This board is ideal for prototyping and developing embedded applications, thanks to its flexibility, standard interfaces (Arduino and Morpho) and ease of programming via ST-Link.

The STM32F401RE is a microcontroller in the STM32F4 series, featuring an ARM Cortex-M4 core with a floating point unit (FPU).

The main features are:

- **Core**: ARM Cortex-M4 a 84 MHz.
- **Flash Memory**: 512 KB.
- **RAM**: 96 KB.
- **I/O Port**: Up to 50 configurable and interruptible GPIOs.
- **Timer**: Multiple advanced and basic timers.
- **Interfaces**: Supporta UART, I2C, SPI e USB 2.0 OTG (On-The-Go).
- **ADC/DAC**: ADC a 12 bit.
- **Energy consumption**: Optimized for low power applications.



STM32 Microcontroller Naming Convention

The *naming convention* for microcontrollers STM32 encodes their characteristics. For the microcontroller **STM32F401RE** these are coded as follows:

- **STM32**: Indicates the family of microcontrollers produced by STMicroelectronics.
- **F**: Microcontroller series. The letter "F" represents high-performance microcontrollers.
- **4**: Specific series, oriented towards advanced performance.
- **01**: Specific model within the F4 series.
- **R**: Indicates the package type (LQFP64 for theNucleoF401RE).
- **AND**: Indicates the size of the Flash memory (512 KB).

Hardware features of theNucleoF401RE

The card NucleoF401RE offers the following hardware features:

- **User LED and Programmable Button:** Status LED (LD2) connected to PA5 pin, can be used to indicate states or signals in the program.
- **User button (USER)**: The blue button is connected to the pin**PC13**and is programmable to interact with the firmware.
- **Reset Button:**The black button allows you to reboot the microcontroller without having to unplug and reconnect the power.
- **Pin Standards: Arduino and Morpho:**
    - **Arduino Uno R3 Standard**: The board is compatible with Arduino expansion boards, thanks to the standard pin layout. This simplifies integration with existing sensors and modules.
    - **ST Morpho Connectors**: In addition to the Arduino layout, additional pins are available via the Morpho connectors, allowing access to all the features of the microcontroller, such as additional GPIOs, ADCs and serial interfaces.
- **ST-Link Integrated:**The board comes with a programmer/debugger **ST-Link/V2-1** integrated, eliminating the need for external devices. This allows:
    - Firmware programming.
    - Debug tramite **SWD** (Serial Wire Debug).
    - A virtual USB interface for serial communication (**VCP**).

The NucleoF401RE is a powerful and versatile platform for embedded development. The combination of the microcontroller **STM32F401RE** with standard interfaces like Arduino and Morpho, the built-in ST-Link and easily accessible peripherals make it suitable for both beginner and advanced projects.

Pin labels

In the context of electronics and microcontroller development, pin labels are identifiers used to describe the function or purpose of individual pins on a microcontroller, integrated circuit (IC), or other electronic component. These labels help users understand the role of the pin and connect it correctly in a circuit.

Common categories of pin labels are:

- **Power and Ground:**
  - **VCC / VDD / VIN:**Power input (positive voltage).
  - **GND:**Ground connection (reference to 0V).
  - **3V3 / 5V:**Outputs or inputs at specific voltages.
- **Digital Input/Output:**
  - **GPIO: General purpose input/output pins.** Often labeled as P0.0, P1.1, etc., or with the port number and pin.
  - **PWM:** Pin capable of generating pulse-width modulated signals.
  - **INT:** Interrupt pins used to detect specific events.
- **Analog Input/Output:**
  - **ADC / A IN:** Input pin for the analog-to-digital converter.
  - **DAC / A OUT:** Output pin for the digital-to-analog converter.
- **Communication:**
  - **UART:**
    - **TX:**Transmission line.
    - **RX:**Receiving line.
  - **I²C:**
    - **SCL:**Serial clock line.
    - **SDA:**Serial data line.
  - **SPI:**
    - **SCK:**Serial clock.
    - **SMOKE:** Master Out Slave In.
    - **MISO:** Master In Slave Out.
    - **CS / SS:** Chip Select / Slave Select.
  - **CAN:**
    - **SOUP:** CAN High.
    - **LIVE:** CAN Low.
- **Functions Speciali (Special Functions):**
  - **RESET / RST:** Pin used to reset the device.
  - **BOOT:** Pin used to configure the boot mode.
  - **CLK:** Input/output pin for the clock.
  - **IN:** Enable pin to turn a device on or off.
  - **LED / IND:** Pin for indicator LEDs.

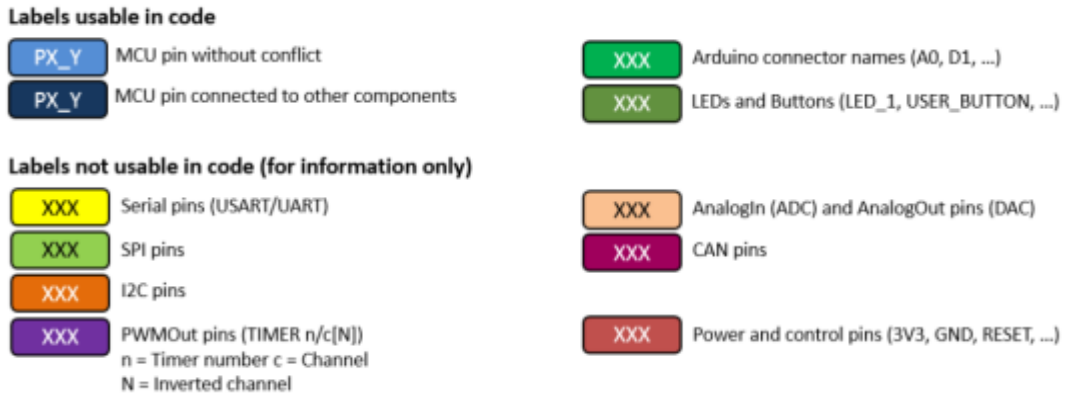
Microcontroller manufacturers generally label pins based on:

- **Functional grouping:** Pins can have multiple functions (for example, a GPIO pin that also acts as an SDA line for I²C).
- **Pin Mapping:** Defined in the microcontroller datasheet, often with diagrams or tables showing possible configurations.

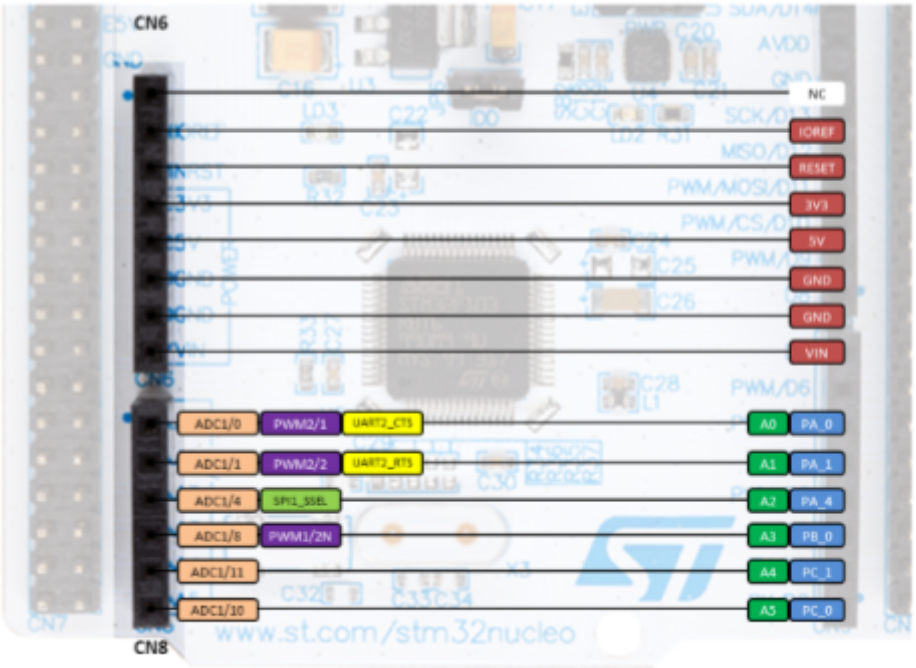
Example:
A pin could be labeled as**P1.0/TXD/PWM1**, indicating that it can function as:

- General Purpose Input/Output (GPIO)**P1.0**.
- UART Transmission (**TXD**).
- PWM output (**PWM1**).

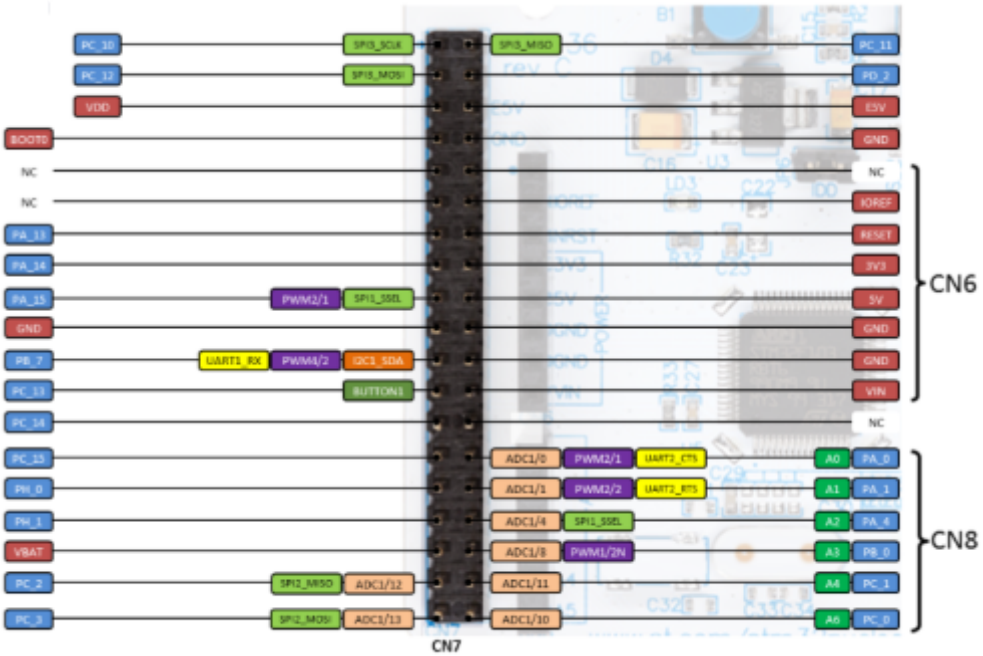
When working with a microcontroller, always refer to the datasheet or pin diagram for an accurate description of pin labels and configurations.



Arduino-compatibile headers



Morpho headers. These headers give access to all pins of the STM32.



CubeIDE

An **Integrated Development Environment (IDE)** is an integrated software development environment designed to simplify writing, debugging, and compiling code. An IDE brings together all the essential development tools in a single graphical interface, including:

- **Source code editor**: To write and edit code.
- **Compiler/Linker**: To translate the code into a language that the microcontroller can understand.
- **Debugger**: To find and correct errors in the code.
- **Simulators or programming tools**: To test the code directly on real hardware.

Using an IDE saves developers time and reduces errors, as it eliminates the need to manage separate tools.

STM32CubeIDE is the official development environment of **STMicroelectronics** designed for programming and debugging STM32 microcontrollers. Based on Eclipse and integrated with GCC (GNU Compiler Collection),STM32CubeIDE combines advanced tools to design and implement embedded applications.

The main features ofSTM32CubeIDE I am:

- **Integration withSTM32CubeMX**: STM32CubeIDE includes **STM32CubeMX**,A powerful graphical configurator that allows you to configure the pins, peripherals and clocks of the microcontroller through a visual interface. It automatically generates the initialization code, reducing the time required to configure the microcontroller.

- **Advanced Debugging Support**: Compatible with hardware debuggers such as **ST-Link** and supports advanced features such as breakpoints, watchpoints and real-time variable analysis.
- **Multi-Toolchain Support**: It uses GCC as the default compiler but supports other toolchains, offering flexibility in development.
- **Firmware Management**: Allows you to download and update STM32 firmware packages directly from the interface, ensuring your projects are always up to date.
- **Cross-Platform**: Available for Windows, macOS and Linux, making it accessible to a large community of developers.

STM32CubeIDE It is ideal for developers of all levels, thanks to its intuitive interface and deep integration with the STM32 microcontroller family. It is particularly useful for those looking for:

- A platform all in one for embedded programming.
- Powerful tools for both simple and complex projects.
- Native support for a wide range of cardsSTM32.

Installation

System Requirements

Before you begin, make sure your computer meets the minimum requirements:

- **Operating system**:
  - Windows 10 o or later (64-bit).
  - macOS 10.15 or later.
  - Linux distributions include Ubuntu 18.04/20.04/22.04 (64-bit).
- **RAM**: At least 4 GB (8 GB or more recommended).
- **Disk space**: About 1.5 GB for installation.
- **Java**: It is not required, STM32CubeIDE includes the necessary Java runtime.

Download STM32CubeIDE

- Go to the official STMicroelectronics website: https://www.st.com/en/development-tools/stm32cubeide.html
- Select your platform (Windows, macOS, Linux) and click on **Download**.
- **Registration (optional)**: You may be asked to create an account or log in to download the software.

Installation on Windows

- **Run the installation file** downloaded (`STM32CubeIDE-Win64-<version>.exe`).
- Follow the steps in the installation wizard:
  - Accept the terms and conditions.
  - Choose the installation directory (e.g.:`C:\STMicroelectronics\STM32CubeIDE`).
- Select add-ons (optional):
  - **OpenOCD support**.
  - **Driver ST-Link**.
- Click on **Install** and wait for completion.

Installation on macOS

- **Run the downloaded DMG package** (`STM32CubeIDE-<version>.dmg`).
- Drag the app STM32CubeIDEin the folder**Applications**.
- If prompted, allow apps downloaded from identified developers to run:
  - Go to**System Preferences > Security & Privacy > General**.
  - Click on **Consented**.

Installation on Linux

- **Make the downloaded file executable**:
  chmod +x STM32CubeIDE-<version>.linux64.run
- **Run the installation**:
  ./STM32CubeIDE-<version>.linux64.run
- Follow the instructions in the terminal to complete the installation.
- **Add USB drivers udev rules** per ST-Link:
  - Download the rules from: ST-Link udev rules.
  - Copy the file `.rules` in the directory`/etc/udev/rules.d/`.
- Reload the rules:
  - sudo udevadm control --reload-rules

Configuring theSTM32CubeIDE

- **StartSTM32CubeIDE**:
  - Windows: Click on the Start menu icon.
- macOS/Linux: Find the appSTM32CubeIDEin applications or run the command:
  ./STM32CubeIDE
- **Set the working directory (workspace)**:
  - Choose a folder for your projects (e.g.:C:\WS_CoreOfEmbeddedSystems).

Introduction toSTM32CubeMX

STM32CubeMXis a powerful software tool provided by STMicroelectronics to configure, initialize and generate code for STM32 microcontrollers. It can be used either as a stand-alone application or as an integrated module in the development environment**STM32CubeIDE**.

The main features are:

- **Microcontroller graphical configuration**: Allows you to configure pins and peripherals through a user-friendly graphical interface.
- **Clock Configuration**: Facilitates setting of system clocks (e.g. HSE, LSE, PLL) with clear visual representation.
- **Automatic code generation**: Produces C code to initialize peripherals and configurations.
- **Middleware support**: Configure and generate code for libraries like FreeRTOS, FatFS, USB and more.
- **Energy consumption simulation**: Includes tools to estimate the microcontroller's power consumption in different configurations.

STM32CubeMXIt is available in two modes:

- **Tool Stand-Alone**: Free to download from the official STMicroelectronics website. Ideal for those who want to generate code and use it in different development environments (e.g. Keil, IAR).
- **Integrated module inSTM32CubeIDE:** In this mode,STM32CubeMXIt is perfectly integrated into the development environment, allowing you to quickly move from hardware configuration to code writing and debugging.

Chapter 3 GPIO output

The **GPIO** (General Purpose Input/Output) is one of the most important peripherals of a microcontroller. GPIO pins can be configured as digital inputs or outputs, making them extremely versatile for interfacing with the outside world. In this introductory project, we will learn how to configure a GPIO pin as a digital output to flash the board's green LEDNucleoF401RE usingSTM32CubeIDE.

Project Objectives

- Understand how a GPIO pin works as a digital output.
- Set up a basic project inSTM32CubeIDE.
- Check the green LED of theNucleoF401REusing C code.
- Introduce the concept of time delay to generate a visible blinking.

Basic Theory: What is a GPIO?

GPIOs are configurable pins that can operate in the following modes:

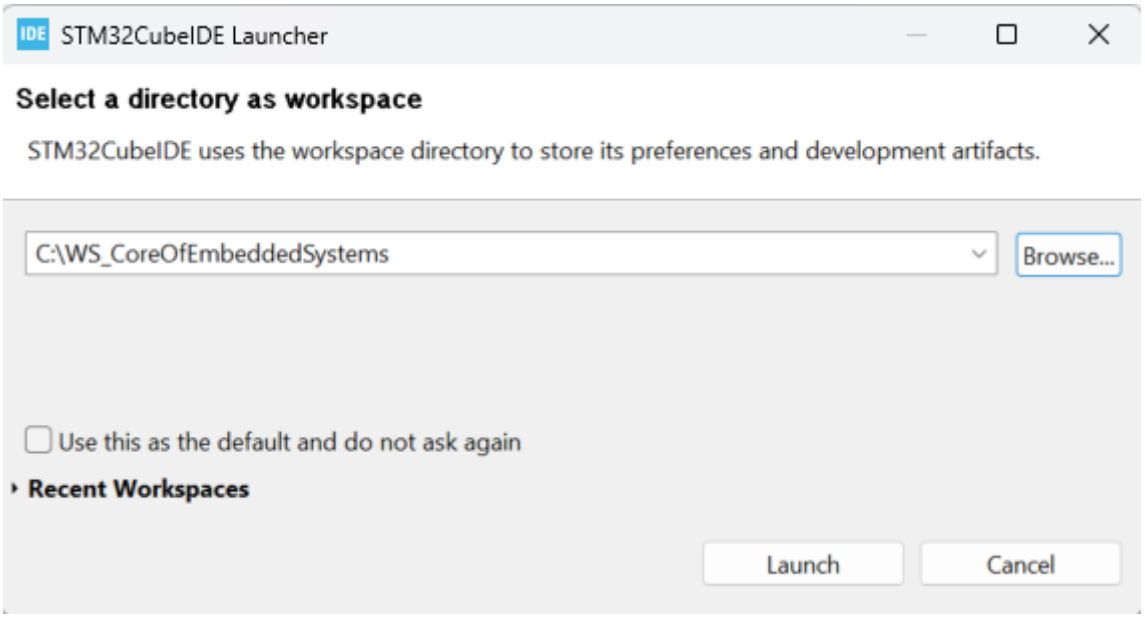
- **Input digital:**To read the logical state (high or low) of an external signal.
- **Output digital:**To generate a logic state (high or low) and control external devices such as LEDs or relays.

When a GPIO pin is configured as a digital output:

- A state **high** typically corresponds to a positive voltage (e.g. 3.3V).
- A state **low** corresponds to a voltage close to 0V.

Setting Up the Project

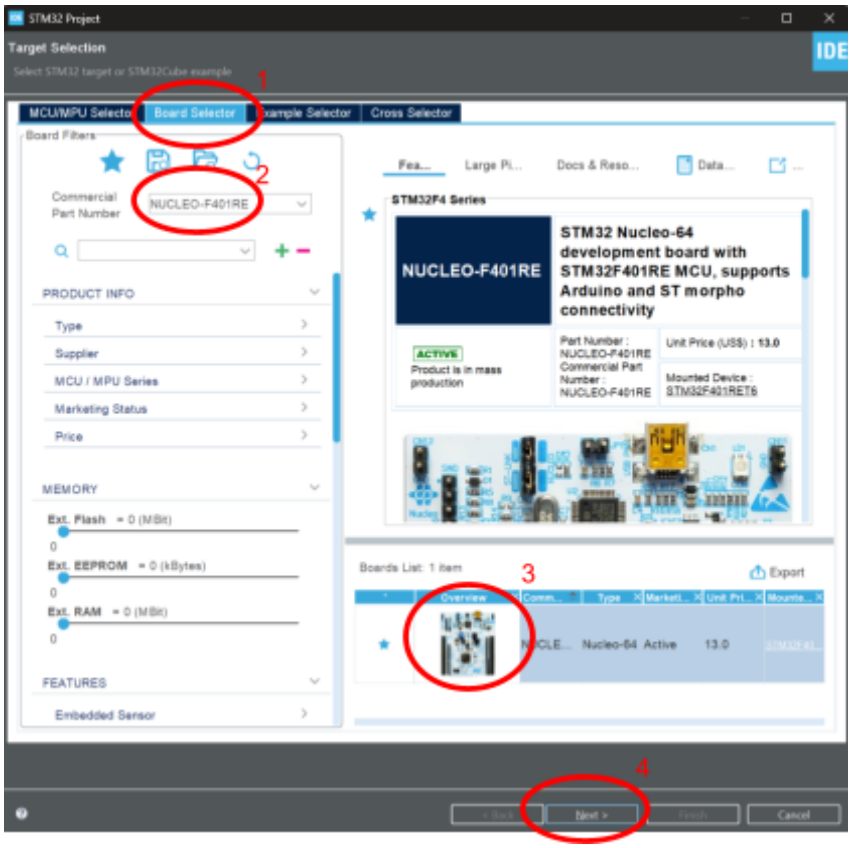
Open STM32CubeIDE and select the previously created workspace



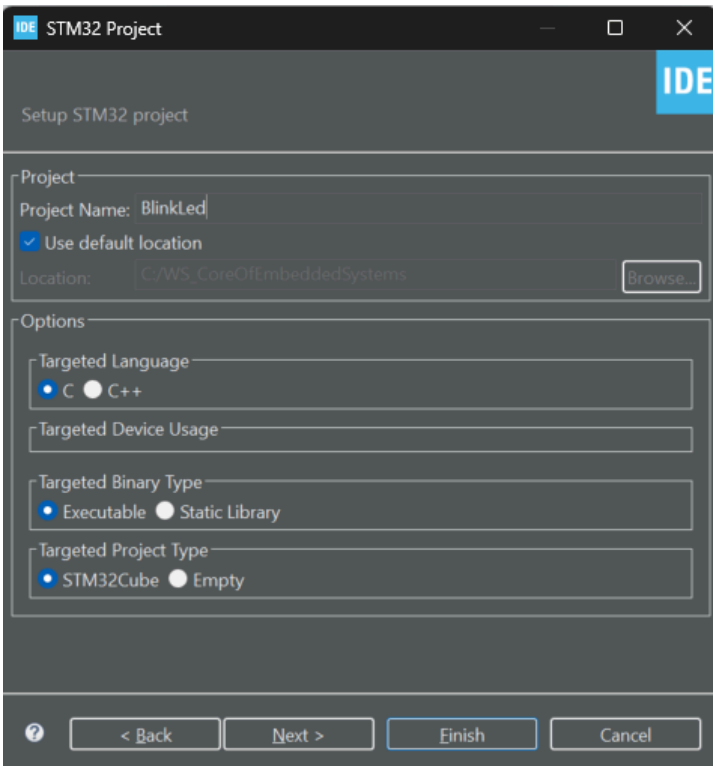
Create a new project in *File/New/STM32 Project.* At this stageCubeIDE,if necessary, it will perform an update

Select *Board Selector*, the card **Nucleus-F401RE** in the field*Commercial Part Number*, and how item in the *Board List*. Finally the button*Next*.

In this way we have chosen the Target tab of the project.

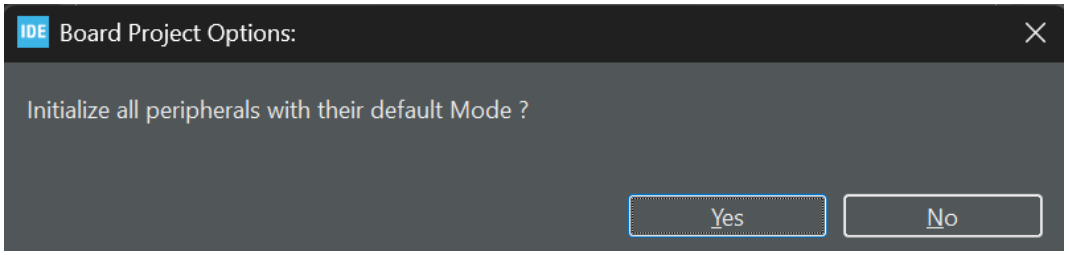


A window will open for the *Setup STM32 project*. Let's enter the name of the project linked leaving everything else unchanged.



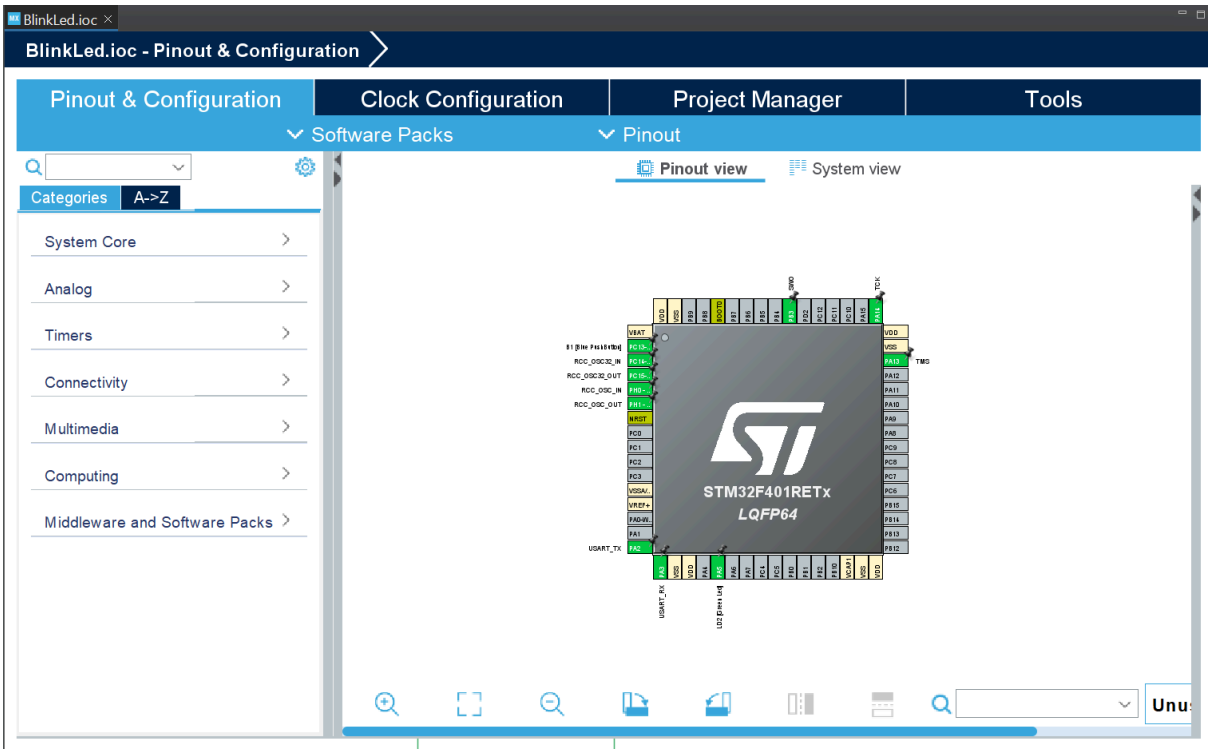
Without having to define anything else, we can close the setup by clicking the button *Finish*.

Let the software initialize the devices in default mode with a click on *Yes*.



By clicking *Yes* the project generation starts. When the generation is complete, a graphical interface will open in the center of the development environment, allowing the configuration of the microcontroller, the Pinout, the Clock and the project. Tools regarding consumption and design are also available.

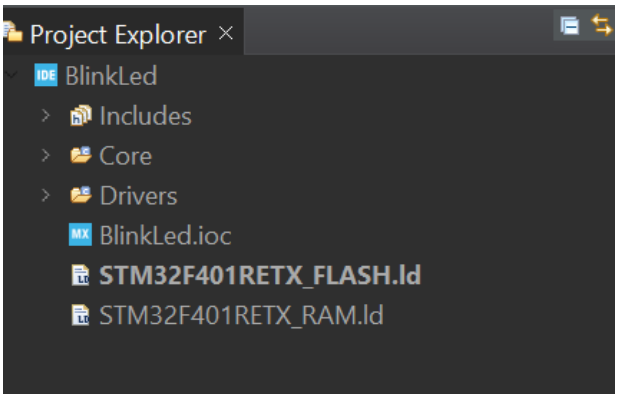
In this manual we will deal exclusively with the Pinout configuration, leaving further details to the reader.



The configurations that we will make will always be editable as they are saved in the file*BlinkLed.ioc*.

Before proceeding with the configuration of the GPIO pins, let's look at the structure of a GPIO project.CubeIDE.This is the same for all projects.

On the left we find the *Project Explorer*



In the context of a typical project STM32CubeIDE, the folders includes, *Core*, and *Drivers* are essential for organizing the source code and project resources.

The Folder ***Includes*** Andan alias or shortcut that collects the paths of the headers (files).h) used in the project. It does not contain files directly but is used to organize the development environment.

The typical content: are links to files .h from other directories, such as those automatically generated by STM32CubeMXor included in the folder*Drivers*. Makes it easier to include

header files in your code without having to specify complex paths. For example, you can write`#include "stm32f4xx_hal.h"`instead of specifying the full path.

The folder ***Core*** contains the main project files, both automatically generated by STM32CubeMX that are written by the user. This folder is crucial for the application to function.

The main sub folders are:

- **`Inc/`**: contains header files (`.h`) project specific such as the file`main.h`, which includes global definitions, macros, and variable or function declarations.
- **`Src/:`** contains iSource files (`.c`) that include the main code of the project, such as the File`main.c`, where the program entry point and application logic resides. Other files may include specific implementations, such as callbacks or interrupt handlers.
- **`Startup/:`** contains the startup file of the microcontroller (`startup_stm32xxx.s`), written in assembly, which handles system initialization (e.g. stack, interrupt vectors).

The folder `Core` is where you write and organize custom code for your application.

The folder Drivers contains the low-level (LL - Low Level) and hardware abstraction layer (HAL - Hardware Abstraction Layer) libraries provided by STM32Cube for interacting with the microcontroller peripherals.

The main subfolders are:

    - **`CMSIS/:`**
        - **CMSIS (Cortex Microcontroller Software Interface Standard)**: Standard libraries provided by ARM for accessing low-level registers and for Cortex core-specific operations.
        - Typical files:`core_cm4.h`, `system_stm32xxx.c`, `stm32f4xx_it.c`.
    - **`STM32XXxx_HAL_Driver/:`**
        - Contains HAL (Hardware Abstraction Layer) driver files to handle peripherals like GPIO, UART, SPI, ADC, etc.
        - Typical files:`stm32f4xx_hal_gpio.c`, `stm32f4xx_hal_rcc.c`, etc.
    - **BSP/(optional)**:
        - **Board Support Package**: Contains specific drivers for STM32 development boards (e.g. Nucleo, Discovery).

The files in this folder are used to simplify the configuration and control of hardware devices.

This organization allows the project to be kept modular, scalable and readable, promoting a clear workflow and separation between hardware configurations, libraries and application code.

Compile the project

Compilation is the process by which source code written in a programming language (for example, C or C++) is translated into machine language that the computer or microcontroller can understand. This machine language consists of binary instructions that the processor can directly execute.

Compilation consists of several stages, which may vary slightly depending on the compiler used, but generally include:

- **Preprocessing (Pre-processing)**: The preprocessor analyzes the source code before the actual compilation.
    - Performs operations such as:
        i. Macro expansion.
        ii. Including header files (`#include`).
        iii. Handling conditional directives (`#ifdef, #endif`).
    - The result is a preprocessed source file.
- **Parsing and Syntactic Analysis**: The compiler analyzes the source code to check its syntactic correctness (conformity to the rules of the language).
- **Translation in Intermediate Code**: The source code is translated into an intermediate representation (e.g.,*assembly*) per to make it easier optimization and subsequent translation.
- **Optimization**: The compiler applies techniques to make the code more efficient in terms of speed or memory consumption.
- **Object Code Generation**: The optimized intermediate code is translated into object **code** (file `.thethe .obj`), which contains machine instructions but is still incomplete.
- **Linking (Connection)**: Object files are combined with any external libraries to produce a complete executable file (e.g., a`.elf`, `.binthe.exe`).

Compilers can be:

- **Native compilers**: They create machine code that can be run directly on the system hardware (e.g. GCC for ARM or x86).
- **Cross-platform compilers (Cross-compiler)**: They generate executable code for a different architecture than the one of the system in use (e.g. ARM GCC for STM32 microcontrollers).

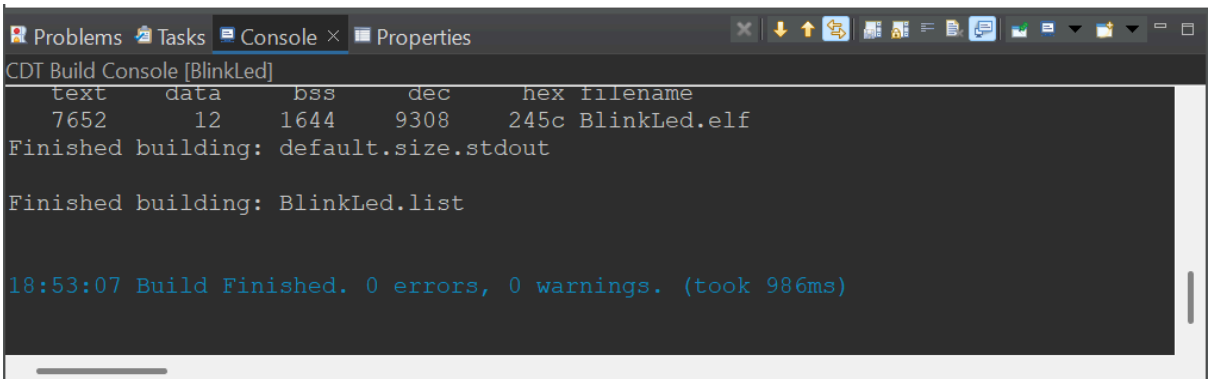
In STM32CubeIDE,Compilation converts source code written in C/C++ (e.g.,main.c) in a binary file (`.binthe.elf`) which can be loaded and run on an STM32 microcontroller.

The generated project does nothing, but it is already compilable and you can download the binary file into the microcontroller's memory.

Let's see how to proceed to perform these two operations.

To fill it out, simply click the hammer icon.

The compilation process generates messages in the*Console*which ends with the printing of the number of errors and the warning.

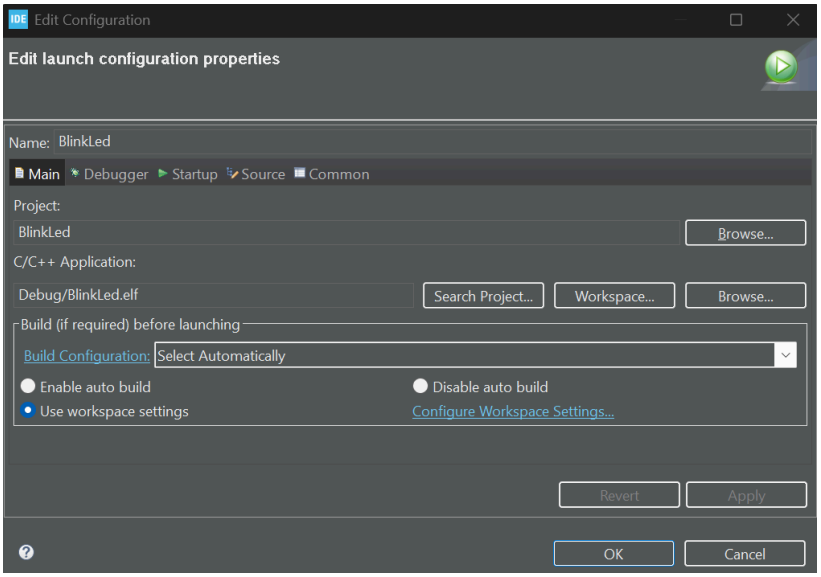


If the project generation process worked correctly, the build completes successfully without errors.

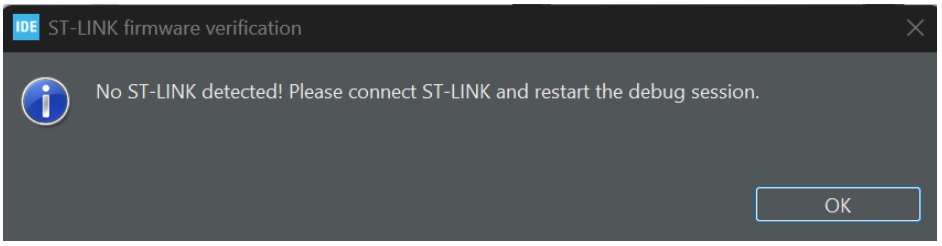
Microcontroller Programming

To program the Nucleo board with the code just compiled, simply click the triangle icon

The first time you need to accept the default configuration by clicking on *OK*



If the Nucleo card is not connected to the PC, a popup will inform us of the absence of ST-Link.

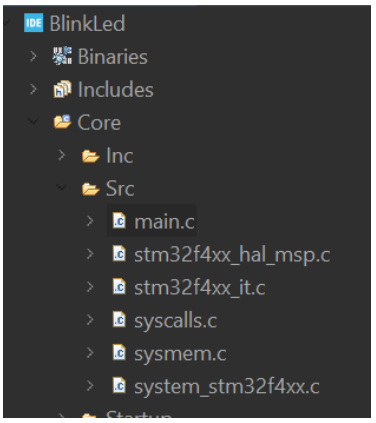


In this case it will be necessary to connect the Nucleo card to the PC and repeat the operation.

He files main.c

The file`main.c`, generated byCubeMXfor a cardNucleoF401RE,is the main file of a HAL (Hardware Abstraction Layer) based project for the STM32F401RE microcontroller. It contains the initial configuration and the main firmware cycle.

This file is located in*Core/Src*



The file includes various headers needed for the code to work:

- `stm32f4xx_hal.h`: Provides definitions and APIs for the HAL library.
- Specific headers for configured devices (e.g.`gpio.h`, `usart.h`, `tim.h`).

Inside it there are declarations of configuration functions such as:

- `void SystemClock_Config(void);` That Configure the system clock.
- `void MX_GPIO_Init(void);`what's uponfigura i pin GPIO.
- Other specific functions to configure peripherals (e.g. UART, Timer).

The main function `main()` is the entry point of the program. This invokes:

- l'HAL initialization,`HAL_Init();`
- The configuring the system clock in the function`SystemClock_Config()`.
- The Initialization Of Peripherals. Calls initialization functions generated by CubeMX, such as MX_GPIO_Init(); and`MX_USART2_UART_Init();`
- ilt also includes the Main Loop (while):

```
while (1)
{
// Custom functions
}
```

CubeMX includes comments to define where to write your code:

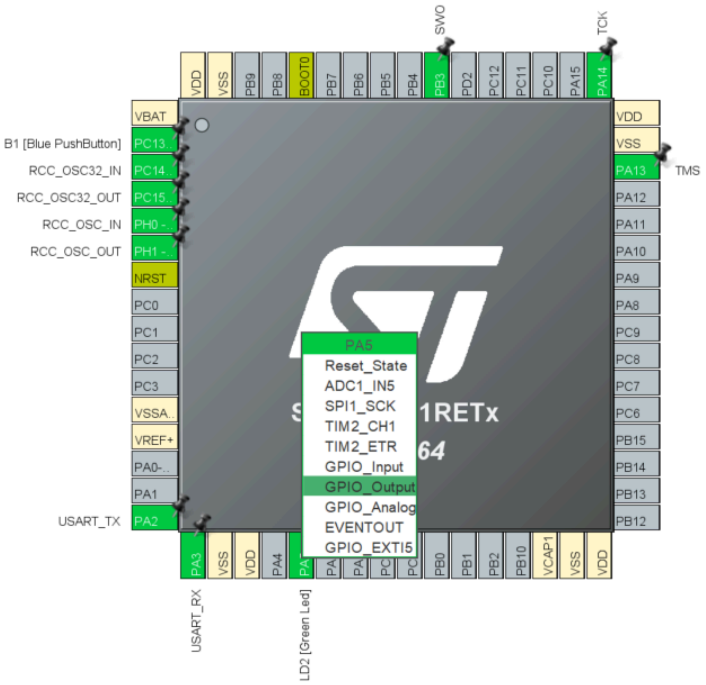
- `/* USER CODE BEGIN 0 */`and`/* USER CODE END 0 */`to define global variables.
- `/* USER CODE BEGIN WHILE */`and`/* USER CODE END WHILE */`to insert custom code into the main loop.

Basically, the file `main.c`It is an automatically generated starting point, structured to be extensible. Developers add their own code in the designated sections without altering the automatically generated parts, making it easy to regenerate inCubeX without losing any changes. The file includes the necessary header files and incorporates all the initialization functions that are called in the main() function.

Pin Configuration

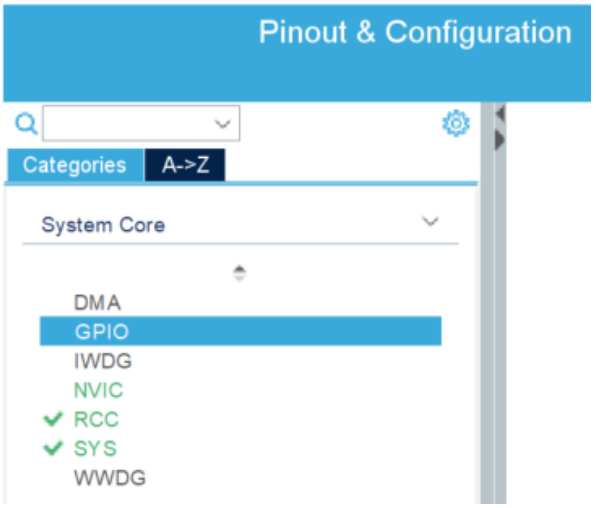
In the microcontroller view identify the pin connected to the green LED. For the Nucleo-F401RE, the green LED is connected to the pin PA5.

CubeMX should have configured the pin as a GPIO output. If the pin is green it is already configured. However we can proceed as follows to analyze the configuration:

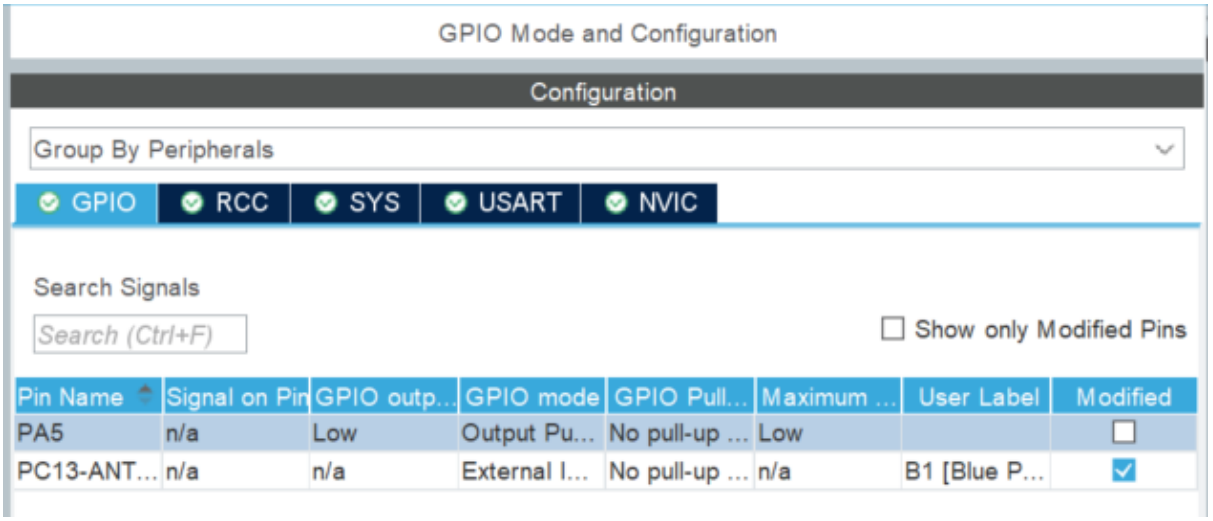


Click on PA5 and select **GPIO_Output** among all the possible *alternate functions available* for pinning.

In *System Core*select GPIO



This will display all pins configured as GPIO.

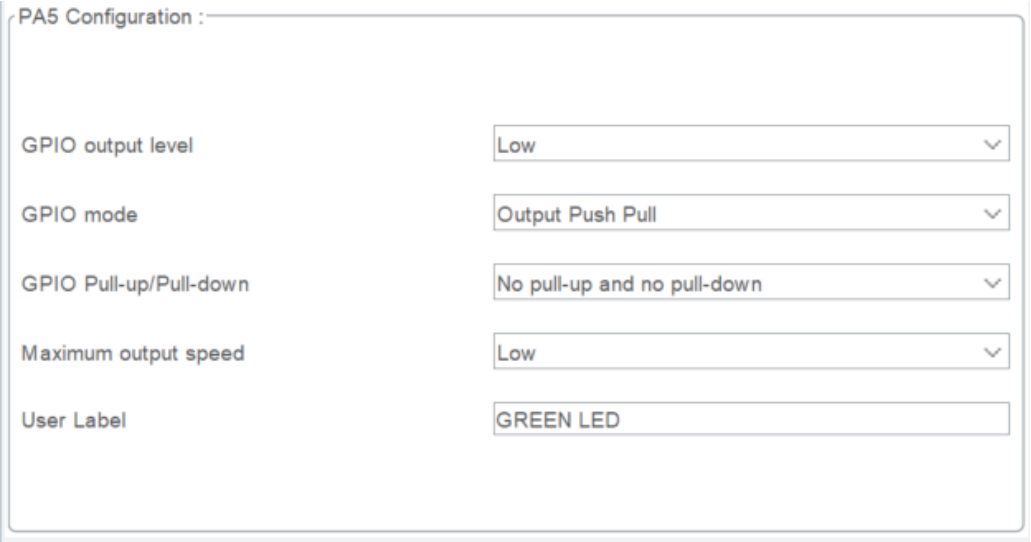


He pin **PA5** (Green LED) in the list, should be configured as follows:

- **Mode**: Output Mode
  i. In a GPIO configured in mode push-pull, the pin can actively drive both the high logic state (by connecting to the supply voltage) and the low logic state (by connecting to GND). In other words, the microcontroller internally supplies current both when the pin is high and when it is low. This configuration is typically the default for standard digital outputs, because it ensures transitions fast and does not require external components.
  ii. In a GPIO configured in mode open-drain, instead, when the internal transistor is "open" the pin is disconnected (high impedance) and does not supply any voltage, while when the transistor is "closed" the pin is connected to ground (GND). In order to have a high level onan exit open-drain, an external pull-up resistor (or internal, if available in the microcontroller) is required.

- **Pull-up/Pull-down**: No Pull (usually not necessary, but depends on the specific design)
- **Speed**: Low, Medium, High (choose according to your needs; for an LED it is generally sufficient**Low**)
- **Output Type**: Push-Pull

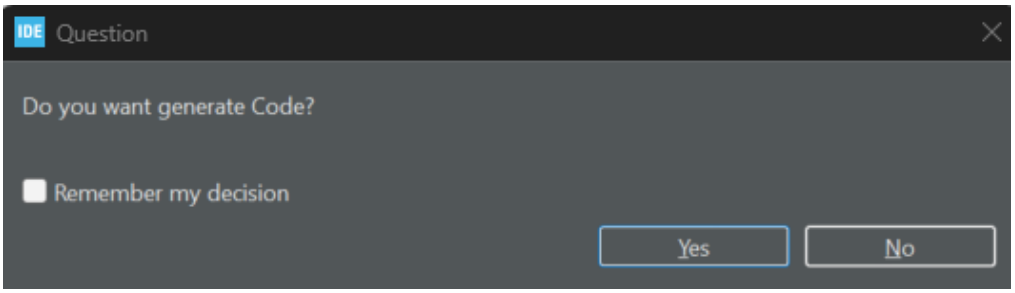
By clicking on the PA5 pin you can open the configuration window.



For this project the default configuration values are fine. We can define a label *User Label That* helps to identify the functionality of the pin. This can be done for all pins, regardless of functionality.

With these steps, the PA5 pin has been configured as a GPIO output pin and we can start using it.

By saving the project, CubeMXasks us if we want to generate the code.



We accept the generation with one click on*Yes*. Popping up *Remember my decision*, code generation will be performed on every save.

The PA5 pin configuration, having been defined as a pin GPIO_Output will be present in the function `static void MX_GPIO_Init(void);`

Aspetti del Linguaggio C

- **static:** Restricts the function's visibility to the current file, preventing name conflicts with similar functions in complex projects.
- **void:** Indicates that the function accepts no parameters and does not return a value.

Let's look at the code of the initialization function generated by CubeMX.

```
GPIO_InitTypeDef GPIO_InitStruct = {0};
```

- Create a variable of type `GPIO_InitTypeDef`, used to configure GPIO pin parameters—such as mode, speed, output type, etc.
- Initialize the structure to zero to ensure default values.

```
__HAL_RCC_GPIOC_CLK_ENABLE();

__HAL_RCC_GPIOH_CLK_ENABLE();

__HAL_RCC_GPIOA_CLK_ENABLE();

__HAL_RCC_GPIOB_CLK_ENABLE();
```

- These macros enable the clock for the various GPIO ports (C, H, A, B).
- Without clock enablement, the GPIOs will not function.

```
HAL_GPIO_WritePin(GREEN_LED_GPIO_Port,                GREEN_LED_Pin,
GPIO_PIN_RESET);
```

- Set the initial level of the `GREEN_LED` pin to `RESET` (LOW).
- `GREEN_LED_GPIO_Port` and `GREEN_LED_Pin` are macros automatically defined by STM32CubeMX to identify the port and pin associated with the LED.

```
GPIO_InitStruct.Pin = GREEN_LED_Pin;

GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;

GPIO_InitStruct.Pull = GPIO_NOPULL;

GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

HAL_GPIO_Init(GREEN_LED_GPIO_Port, &GPIO_InitStruct);
```

- **Pin:** Specifies the green LED pin.
- **Mode:** Configures the pin as push-pull output (OUTPUT_PP).
- **Pull:** No pull-up or pull-down resistor activated.
- **Speed:** Sets the pin speed to LOW (sufficient for an LED).
- **HAL_GPIO_Init:** Initializes the pin with the defined parameters.

Flashing LED

Having completed the configuration of the PA5 pin to which the green LED is connected, we can proceed with the implementation of the logic.

To flash an LED, we toggle the state of the GPIO pin between high (HIGH) and low (LOW).

A delay is needed between transitions to make the flashing visible.

With this logic we must follow the following steps:

- You write **1** on the register to turn on the LED.
- Wait for a certain amount of time (e.g. 500 ms).
- You write **0** on the register to turn off the LED.
- repeat

For the implementation we use the already available functions of the HAL (Hardware Abstraction Layer)

To turn the LED on or off, we use the write function of a pin. This function takes as input the port to which the pin belongs, the pin itself and the state you want to write.

```
HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, GPIO_PIN_SET);
```

When we assign a name to the pin during configuration,CubeMX generates a macro for the port that includes the pin and a macro for the pin. In this case, the label assigned to pin PA5

isGREEN_LED, andCubeMX generated the macro GREEN_LED_GPIO_port which identifies the port and GREEN_LED_Pin which identifies the pin.

The two default macros, GPIO_PIN_SETand GPIO_PIN_RESET respectively raise and lower the output of a digital pin.

Below is the code to insert into the loop in the USER CODE section.

```
/* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
   HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, GPIO_PIN_RESET);
   HAL_Delay(500);
   HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, GPIO_PIN_SET);
   HAL_Delay(500);
 /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
 }
 /* USER CODE END 3 */
```

To insert the delay between switching on and off and vice versa, we use another HAL function.
The function void HAL_Delay(uint32_y delay) takes as input the waiting time expressed in milliseconds represented by a 32-bit integer.

The LED flashing can also be obtained using another HAL function.
The void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin) function changes the state of the digital output pin. If the current state is High, the next state will be Low and vice versa.

Below is the code to insert into the loop in the USER CODE section.

```
/* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
   HAL_GPIO_TogglePin(GREEN_LED_GPIO_Port,
GREEN_LED_Pin);
   HAL_Delay(1000);
   /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
 }
 /* USER CODE END 3 */
}
```

Fill out and download the program the Nucleo board.

Chapter 4 Serial communication

In this project, we will learn how to configure the USART2 peripheral of the STM32F401RE microcontroller in asynchronous serial communication mode to send and receive data from a PC terminal. We will use a terminal (e.g. PuTTY or generic serial monitor) to transmit text commands to the microcontroller and, in response, make the green LED (LD2) on the Nucleo board flash or turn off.

Project Objectives

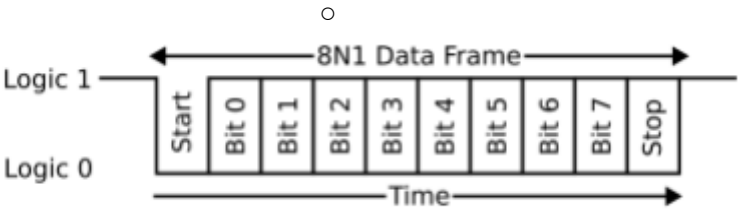
- Understand how a UART/USART serial port works in asynchronous mode.
- Set up a basic project inSTM32CubeIDE with USART2 (PA2=Tx, PA3=Rx).
- Set communication parameters: baud rate,bits of data, parity, stop bit.
- Send strings from the microcontroller to the PC (e.g. welcome message).
- Receive commands from the PC to control the LED (e.g. “ON”/“OFF”).

Basic Theory: Asynchronous Serial Communication (UART)

Universal Asynchronous Receiver Transmitter (UART) serial communication is one of the simplest and most popular methods for making microcontrollers and PCs communicate with each other.

Each byte of data is transmitted as a sequence of bits on the Tx channel and encapsulated in a frame composed of:

- **start bit** (level “0”),
- bo - b7 data **bit**,
- possible parity **bit** (None, Even, Odd),
- 1 or 2 stop **bit** (level “1”).



The transmission speed is measured in bits per second and is called *Baud rate.* (e.g. 9600, 115200 bps). Must be set to the same value on both the transmitter and receiver.
The Tx/Rx lines of the USART2 (hardware) port are assigned to the PA2 (USART2_TX) and PA3 (USART2_RX) pins respectively.

Reception Management Mode

There are two possible ways to receive bytes:

- **Polling:** The CPU repeatedly calls the function `HAL_UART_Receive()`(check the flag `RXNE`) in a loop. Simple to implement, but keeps the CPU busy even when there is no data.

- **Interrupt:** An interrupt is triggered on receipt of each byte (flag `RXNE`), executing a callback (`HAL_UART_RxCpltCallback`). The CPU can perform other tasks and is only interrupted when new data arrives.

Alternatively, for high throughput, i.e. for a large amount of data transmitted, DMA mode can be used for block transfers without continuous CPU intervention. This topic is beyond the scope of this book.

In the following paragraphs we will see step by step how to:

- Enable and parameterizeUSART2 in STM32CubeIDE,
- Write the initialization code in`MX_USART2_UART_Init()`,
- Send a welcome message on boot,
- Manage receiving commands via interrupt to turn the LED on/off.

Setting Up the Serial Polling Project

Open STM32CubeIDE, select the previously created workspace and create a project with the name *Serial_polling*. If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

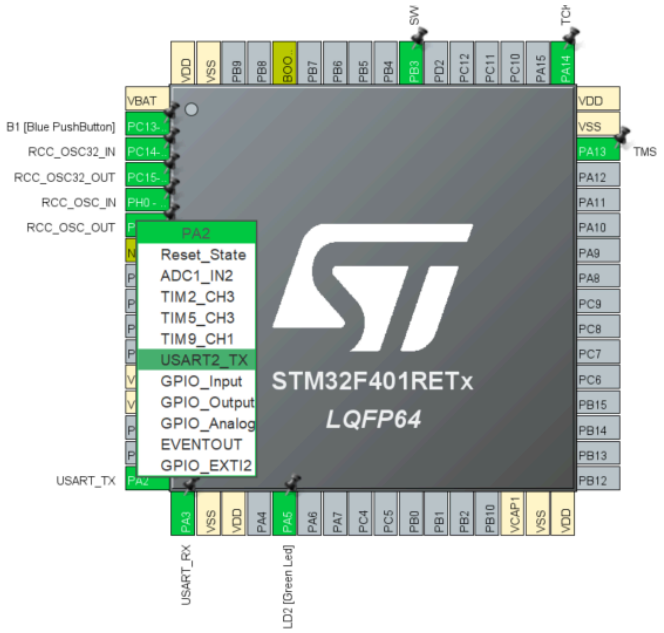
USART2 Pin Configuration

Steps to configure pinsUSART2 (PA2=Tx, PA3=Rx) in STM32CubeMX are as follows:

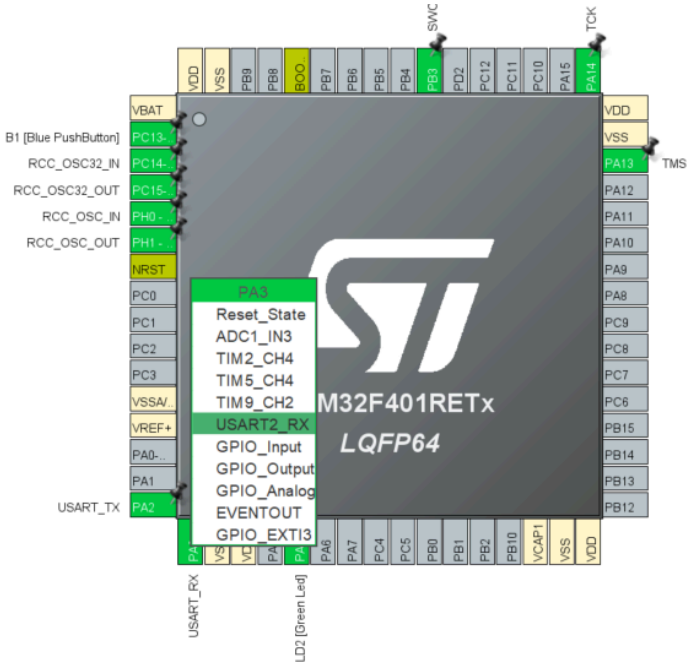
By double clicking on the file *Serial_polling.ioc*, open the configuration file and select *Pinout & Configuration Tool*. Locate the pins associated with the port USART2, PA2 and PA3.

If you chose to perform a default configuration during project creation, the port may already be configured with the pins colored green.

For pin PA2, the alternate function must be selected *USART2_TX*



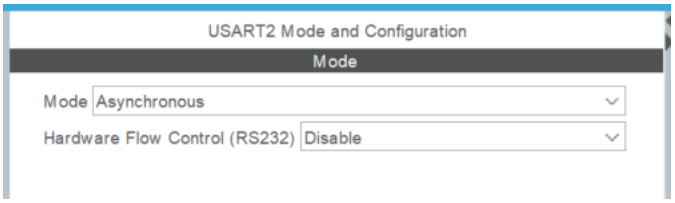
Instead, for the PA3 pin, the alternate function must be selected *USART2_RX*.



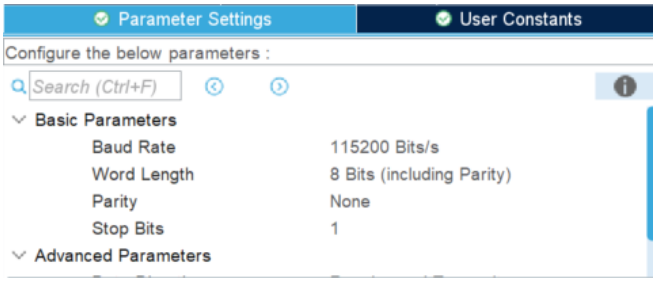
Having configured the alternates function related to the port pins USART2, we can move on to configuring the communication parameters.

In the menu*Connectivity*, we select *USART2*

We choose asynchronous mode without enabling hardware flow control.



Finally, as parameters we select:



Finally, let's configure the LED as seen in the previous chapter.
We save the project and accept the automatic generation of the code

At this point CubeMX will have generated the function `MX_USART2_UART_Init()` and instantiated the handle `huart2` ready to use.

```
UART_HandleTypeDef huart2;
```

We define the message to send to the client and the buffer for receiving the command. We insert the following code in the sections dedicated to the user.

```
/* USER CODE BEGIN 1 */
    const char prompt[] = "Send command: ON or OFF\r\n";
    uint8_t rxBuf[4]; // spazio per "ON\0" o "OFF\0"
/* USER CODE END 1 */
```

We include the library for string handling. We do this in the private includes section

```
/* Private includes
----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <string.h>
/* USER CODE END Includes */
```

In the while loop we insert the following code

```
while (1)
{
    // 1) Send prompt
    HAL_UART_Transmit(&huart2, (uint8_t*)prompt, strlen(prompt), HAL_MAX_DELAY);

    // 2) Receive command in polling (3 characters + terminator)
    //    Here we always wait for 3 bytes; you can adjust to read until '\n'
    memset(rxBuf, 0, sizeof(rxBuf));
    do {
          HAL_UART_Receive(&huart2, &rxChar, 1, HAL_MAX_DELAY);
          if (rxChar != '\r' && rxChar != '\n' && idx < (sizeof(rxBuf) - 1)) {
              rxBuf[idx++] = rxChar;
          }
       } while (rxChar != '\n');
    rxBuf[3] = '\0'; // null-terminator

    // 3) Compare and act on the LED
    if (strcmp((char*)rxBuf, "ON") == 0)
    {
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);   // LED ON
    }
    else if (strcmp((char*)rxBuf, "OFF") == 0)
    {
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET); // LED OFF
    }
    // else: unrecognized command, repeat prompt
}
```

To send the prompt message we use the function `HAL_UART_Transmit()`

Its parameters are:

```
HAL_UART_Transmit(
      &huart2,      // 1) Pointer to the UART handle (here
                                USART2)
      (uint8_t*)prompt,  // 2) Pointer to the data buffer to
                                transmit (cast to uint8_t*)
      strlen(prompt),    // 3) Length of the buffer in bytes
(number
                                of characters in the prompt)
```

```
    HAL_MAX_DELAY      // 4) Maximum timeout in milliseconds
                              (blocks until completion)
);
```

- *&huart2*: indicates which UART device to apply transmission to. The handle contains the settings (baud rate, word length, parity, etc.) configured in MX_USART2_UART_Init().

- *(uint8_t*)prompt*: is the pointer to the text to send, which is originally a C string (char *). The cast to uint8_t* is needed because HAL expects a byte buffer (uint8_t).

- *strlen(prompt)*: calculates how many bytes (characters) there are in the prompt, not counting the terminator '\0'. So the function transmits exactly that many bytes.

- HAL_MAX_DELAY:“infinite” timeout value (effectively blocks the CPU until all bytes have been sent). If transmission is not possible within this time, the function will return an error.

The function `memset()` It is used to “clean” (reset) a memory area before using it, avoiding the presence of “dirty” (residual) values from previous operations.

His signature is:

void * `memset`(void *s, int c, size_t n);

and the parameters are:

- **`rxBuf`**: pointer to the beginning of the array to be reset.
- **`0`**: byte value to write in each position (here zero).
- **`sizeof(rxBuf)`**: number of bytes to write to (the total size of the array).

So:

`memset`(rxBuf, 0, sizeof(rxBuf));

means “fill all the `sizeof(rxBuf)` byte di `rxBuf` with zeros”. In practice you zero the entire buffer`rxBuf`, setting each element to`0`, before receiving the new command via UART. This ensures that, even if the received command is shorter or not finished, the remaining positions remain at zero (`'\0'`), avoiding readings of random values.

This code snippet is for filtering the characters received and to prevent overflow from the buffer `rxBuf` Let's analyze the three conditions:

```
if (rxChar != '\r' // 1) Do not store the carriage return
&& rxChar != '\n' // 2) Do not store the line feed
 && idx < (sizeof(rxBuf)- 1)) // 3) Make sure you leave space
```

```
                                    for the terminator '\0'
{
rxBuf[idx++] = rxChar; // 4) Store the valid character and
                              increase the index
}
```

- **`rxChar != '\r'`** Discard the “carriage return” character (`carriage return`, ASCII 0x0D) sent from many terminals when you press Enter.
- **`rxChar != '\n'`** Discard the “line feed” character (`line feed`, ASCII 0x0A), also generated at the end of the line.
- **`idx < (sizeof(rxBuf) - 1)`** Ensures that the indexid you always stay minor of the buffer length minus one, so as to reserve one byte for the null terminator (`'\0'`) at the end of the string. This avoids writing beyond the bounds of the array (`buffer overflow`).
- **`rxBuf[idx++] = rxChar;`** If all conditions are true, the received character is stored in `rxBuf`to the position `idx` and then `idx` is increased (`idx++`), preparing to write at the next available slot.

In short, this `if` makes sure to save only the significant characters of the command (e.g. “O”, “N”, “F”, “F”) in the buffer, ignoring line terminators, and not to exceed the capacity of the array reserved for the string.

The function call

`HAL_UART_Receive`(&huart2, rxBuf, 3, HAL_MAX_DELAY);

usa l’API HAL in **blocking mode** to read data from the UART peripheral. In detail:

The parameters are

- **`&huart2`**: Pointer to the **handle** of USART2, which contains all the configurations (baud rate, frame formatting, etc.) and status of the device.
- **`rxBuf`**: Pointer to the receive buffer, i.e. the array where the received bytes will be stored. It must be large enough to contain the expected number of bytes.
- **`3`**: Number of bytes to receive. The function will wait until it has received exactly 3 characters (or times out).
- **`HAL_MAX_DELAY`**: Maximum timeout in milliseconds for receiving: using`HAL_MAX_DELAY` The call never times out, blocking execution until the 3 bytes

have been queued by the UART driver.

What happens at runtime?

- The UART driver activates reception and waits for 3 incoming bytes.
- Each time a bit arrives, the hardware reconstructs it into the corresponding byte and stores it in the receive register.
- HAL_UART_Receive transfers each byte from the hardware register to the software buffer (`rxBuf`).
- Only after reading all 3 bytes, the function returns control to the program, which continues executing the next line.

If for some reason the bytes do not arrive (e.g. a disconnected line), the code remains stuck on this call until the data arrives. If you wanted to handle a timeout instead, you could replace `HAL_MAX_DELAY` with an explicit value of milliseconds.

```
rxBuf[3] = '\0';
```

It is used to transform the byte array `rxBuf`in a real “C string” finished from the null character, so that all string manipulation functions (such as `strcmp`, `stren`, etc.) know where to stop.

This code block compares the string received via UART (`rxBuf`) with text commands`"ON"`and`"OFF"`and, based on the result, turns on or off the LED connected to PA5:

```
if (strcmp((char*)rxBuf, "ON") == 0)
{
      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);    // LED
ON
}
else if (strcmp((char*)rxBuf, "OFF") == 0)
{
      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET); // LED
OFF
}
```

- **`strcmp((char*)rxBuf, "ON") == 0`**
  - `Strcmp` compares two C strings character by character.
  - Returns **0** if the strings are exactly the same.
    Here you can check whether `rxBuf` (cast a `char*`) is identical to the string`"ON"`.

- **`HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);`**
  - If the condition is true (received`"ON"`), calls the HAL to bring the PA5 pin high, turning on the LED (macro`GPIO_PIN_SET`).

- **`else if (strcmp((char*)rxBuf, "OFF") == 0)`**
    - If the first condition is false, check if `rxBuf`corresponds to`"OFF"`.

- **`HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);`**
    - If he received`"OFF"`, set PA5 to low level (`GPIO_PIN_RESET`), turning off the LED.

If the content of `rxBuf` is neither`"ON"`nor`"OFF"`, the code enters neither block and simply jumps to the next loop to receive a new command.

Fill out and download the program on the board.

To interact with the Nucleo you need to open a serial client.

Configuring the Serial Interrupt Project

OpenSTM32CubeIDE, select the previously created workspace and create a project with the name*Serial_interrupt*.If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

USART2 Pin Configuration

The port USART2 must be configured as in the previous example. The LED configuration also does not differ from the cases seen previously.

In the case of interrupts, the control over the command is done inside the callback function which is called every time a byte has been received.
The callback function that the programmer must implement is

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
   if (huart->Instance == USART2)
   {
       // 1) Filter out CR/LF and store valid chars
       if (rxChar != '\r' && rxChar != '\n' && idx < (sizeof(rxBuf)-1))
       {
           rxBuf[idx++] = rxChar;
       }
       // 2) If we saw LF, treat as end-of-command
       if (rxChar == '\n')
       {
           rxBuf[idx] = '\0';  // null-terminate
           // 3) Compare and act
           if (strcmp(rxBuf, "ON") == 0)
           {
               HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
           }
           else if (strcmp(rxBuf, "OFF") == 0)
           {
               HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
           }
           // 4) Reset buffer for next command
           idx = 0;
           memset(rxBuf, 0, sizeof(rxBuf));
       }
       // 5) Re-arm the UART receive interrupt for next byte
       HAL_UART_Receive_IT(&huart2, &rxChar, 1);
   }
}
```

Interrupt reception is instead initiated by invoking the HAL function

HAL_UART_Receive_IT(&huart2, &rxChar, 1);

Chapter 5 GPIO input - polling

In this project, we will learn how to configure a GPIO pin as a digital input to read the state of the Nucleo board's user button. We will use the button to turn the green LED on and off..

Project Objectives

- Understand how a GPIO pin works as a digital input.
- Set up a basic project inSTM32CubeIDE.
- Read the status of the blue button NucleoF401RE using C code.

Basic Theory: Reading a GPIO Input

When you intend to use a GPIO pin as a digital input, you must take into account the two reading modes:

- **Polling:** The microcontroller continuously checks the state of the GPIO in a repeating cycle (loop). It is a synchronous method, the CPU is constantly busy reading the pin, even when the state does not change.
- **Interrupt:** The microcontroller runs the main code and "stops" only when the GPIO changes state, generating an interrupt signal. It is an asynchronous method, the CPU takes care of other tasks until notified by the GPIO.

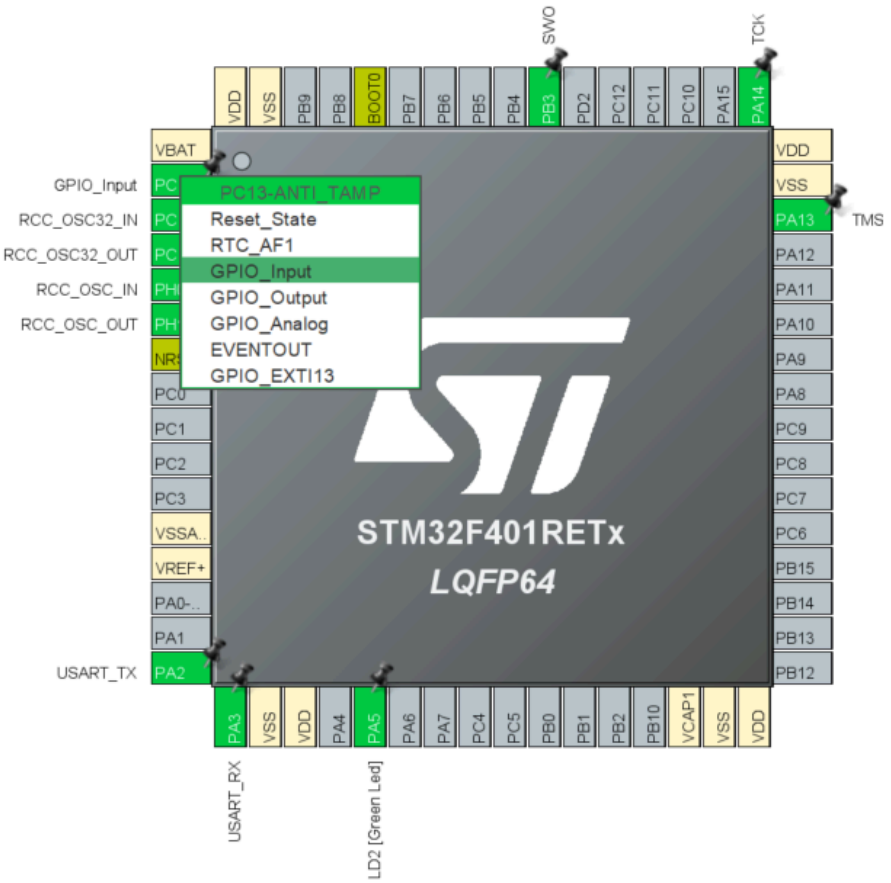
Setting Up the Project

OpenSTM32CubeIDE, select the previously created workspace and create a project with the name*UserButton_polling*. If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

User Button Pin Configuration

Open the Pinout & Configuration Tool and locate the pin associated with the user button on the boardNucleoF401RE,The User Button is connected to the PC13 pin.

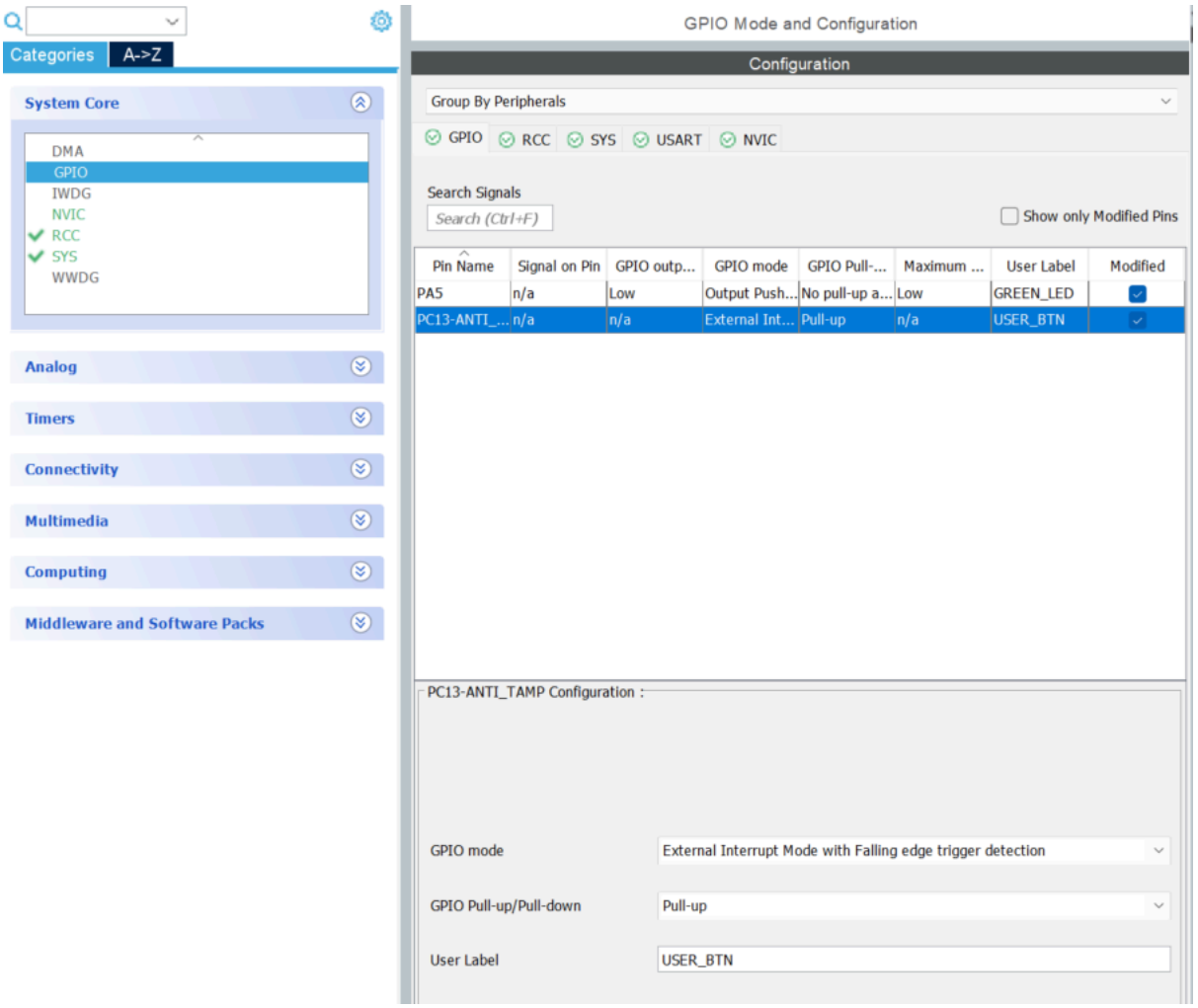
Click on the PC13 pin on the microcontroller map and set it as GPIO_Input.The pin may already have been configured by CubeMX.



If you want to turn on the LED, you also need to configure the corresponding pin as seen previously, assigning the LED pin the label*GREEN_LED*.

In the card **GPIO Configuration**, select the pin **PC13** and configure the following parameters:

- **Mode:** `Input Mode`
- **Pull-up/Pull-down:** `Pull-up`(optional, but often recommended to improve button stability, as the button connects the pin to GND when pressed).
- **Label:** You can optionally assign a label as`USER_BTN`.



Once the pin configuration is complete, we can generate the code. To do this, simply save the project, or in general click on **Project -> Generate Code**.

STM32CubeIDE will generate the code needed to configure the pin in input mode.

Reading the Button

In the file `main.c` (the `main.cpp` for a C++ project), in the main loop (`while(1)`), we read the state of the button using the function `HAL_GPIO_ReadPin`.

When the button is pressed the pin is brought to the low state encoded by the macro GPIO_PIN_RESET. So we will need to introduce a check to verify that the pin has assumed this value.

To turn on the LED, we use the toggle function of the pins associated with the green LED.

Below is the code to insert into the loop in the USER CODE section.

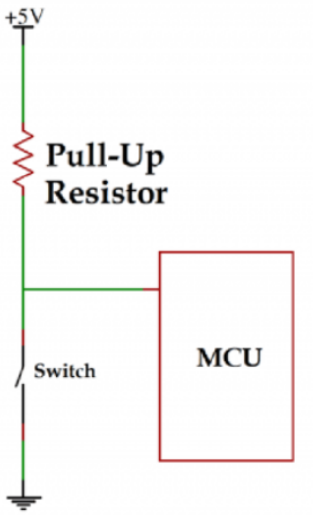
```
/* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
  if (HAL_GPIO_ReadPin(USER_BTN_GPIO_Port, USER_BTN_Pin) ==
GPIO_PIN_RESET){
        HAL_GPIO_TogglePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin);
  }
  HAL_Delay(100);
       /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
 }
 /* USER CODE END 3 */
}
```

This code is for educational purposes only, in practice it should be improved by adding the *debouncing*.
Fill out and download the program on the Nucleo board.

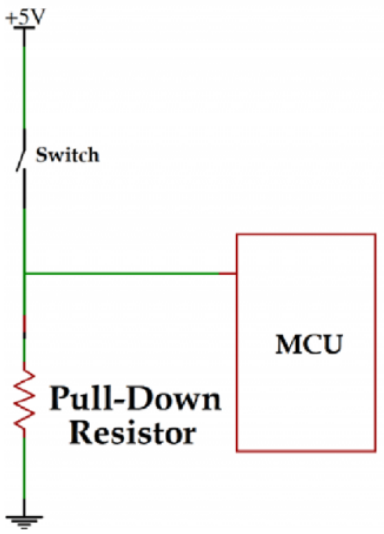
Insight: Pull Up/Down Resistors

Pull-up (or pull-down) resistors are used to ensure that a pin's signal remains stable in a well-defined state (high or low) when it is not actively connected to an external source, such as when a button is not pressed.

- **Behavior without pull-up resistor**: When the button is not pressed, the pin is disconnected (in an "open" circuit) and its logical state is undetermined. This phenomenon is known as **floating state**(floating). A pin in a floating state can detect noise signals or random variations in voltage, leading to unstable or unpredictable readings.
- **Pull-up resistor function**: The pull-up resistor is connected between the input pin and the positive supply voltage (+Vcc). When the button is released (not pressed), the resistor ensures that the pin is "pulled up" to the logic high (HIGH) state. When the button is pressed, it directly connects the pin to GND,forcing it to low logic state (LOW).



- **Button stability**: With a pull-up resistor, the circuit ensures a clear and stable transition between the HIGH (released) and LOW (pressed) states. This eliminates noise and indeterminate states. The same logic applies to the pull-down resistor, which instead "pulls" the pin to GND, ensuring the LOW state when the button is not pressed.



Many modern microcontrollers, such as the STM32, offer internal pull-up/pull-down resistors that are configurable via software.

Chapter 6 GPIO input - interrupt

In this project, we will learn how to configure a GPIO pin to generate a digital interrupt in response to pressing the user button on the Nucleo board. We will use this interrupt to turn the green LED on and off responsively and efficiently.

Project Objectives

- Understand how interrupts associated with a GPIO pin work.
- Set up a basic project inCubeIDEwith interrupt handling.
- Use an interrupt to detect the blue button being pressed NucleoF401RE and handle events using C code.

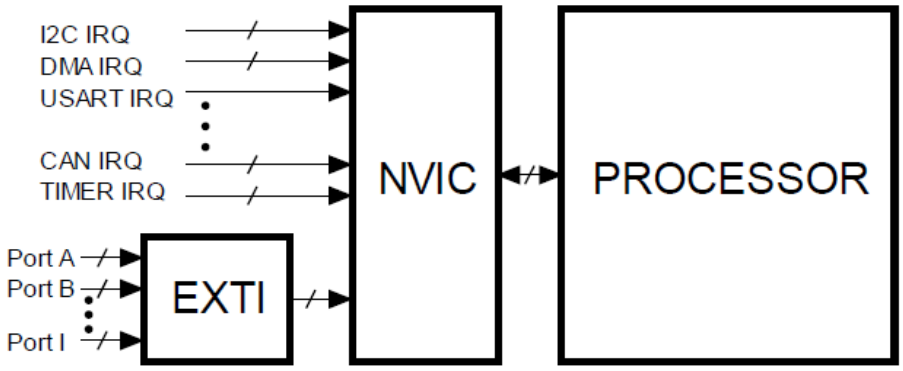
Basic Theory: Generating an Interrupt

An interrupt is a signal that allows the microcontroller to stop the execution of the main code to respond to a specific event, such as the change of state of a GPIO pin. This approach is asynchronous, the CPU can continue to perform other tasks until it is notified of the event via the interrupt.

In the STM32F401RE microcontroller, interrupt management is entrusted to the **NVIC (Nested Vectored Interrupt Controller)**, which prioritizes and dispatches interrupts. When a GPIO pin configured to generate an interrupt detects a change in state (for example, a rising or falling edge), the microcontroller:

- **Detects the event**: The GPIO system registers the state change configured as a trigger (e.g. rising or falling edge).
- **Send the request to NVIC**: The GPIO notifies the NVIC that an interrupt event has occurred.
- **Executes the ISR (Interrupt Service Routine) routine**: The CPU temporarily suspends the main code to execute the associated functional interrupt (ISR), which manages the event.
- **Return to main code**: After completing the ISR, the CPU resumes where it left off.

This system ensures fast response to events without compromising overall efficiency, since the CPU does not have to continuously check the GPIO state (polling method).



The diagram in the figure represents the flow and management of interrupts in STM32 microcontrollers, using the main blocks involved:get outNVIC, and the PROCESSOR.

- EXTI (External Interrupt/Event Controller): This block is responsible for handling interrupts generated by the pin GPIO Interrupt signals that can come from various *ports* (is.Port A, Port B, up to Port I), or from the pins configured to generate interrupt events. The task of the 'EXTI is to identify the pin that generated the interrupt and notify theNVIC.
- NVIC (Nested Vectored Interrupt Controller): This is the main component that manages and prioritizes all system interrupts. It receives requests bothdall'EXTI(external interrupts) or from other microcontroller peripherals, such as:
    - **I2C IRQ**: Interrupt generated by the I2C module.
    - **DMA IRQ**: Interrupt related to direct memory transfers (DMA).
    - **USART IRQ**: USART serial communication module interrupt.
    - **CAN IRQ**: CAN communication module interrupt.
    - **TIMER IRQ**: Interrupt generated by timers.
- The NVIC assigns a priority to interrupts, to ensure that the most critical ones are handled first. It can also handle nested interrupts, that is, suspend an ISR (Interrupt Service Routine) to execute another interrupt with a higher priority.
- PROCESSOR: This is the core of the microcontroller, responsible for executing the main code. When the NVIC notifies an interrupt, the PROCESSOR temporarily suspends the normal flow of the program and switches to executing l'InterruptService Routine (ISR), or the interrupt handling function. Once finished the ISR, the PROCESSOR resumes the main code from where it left off.

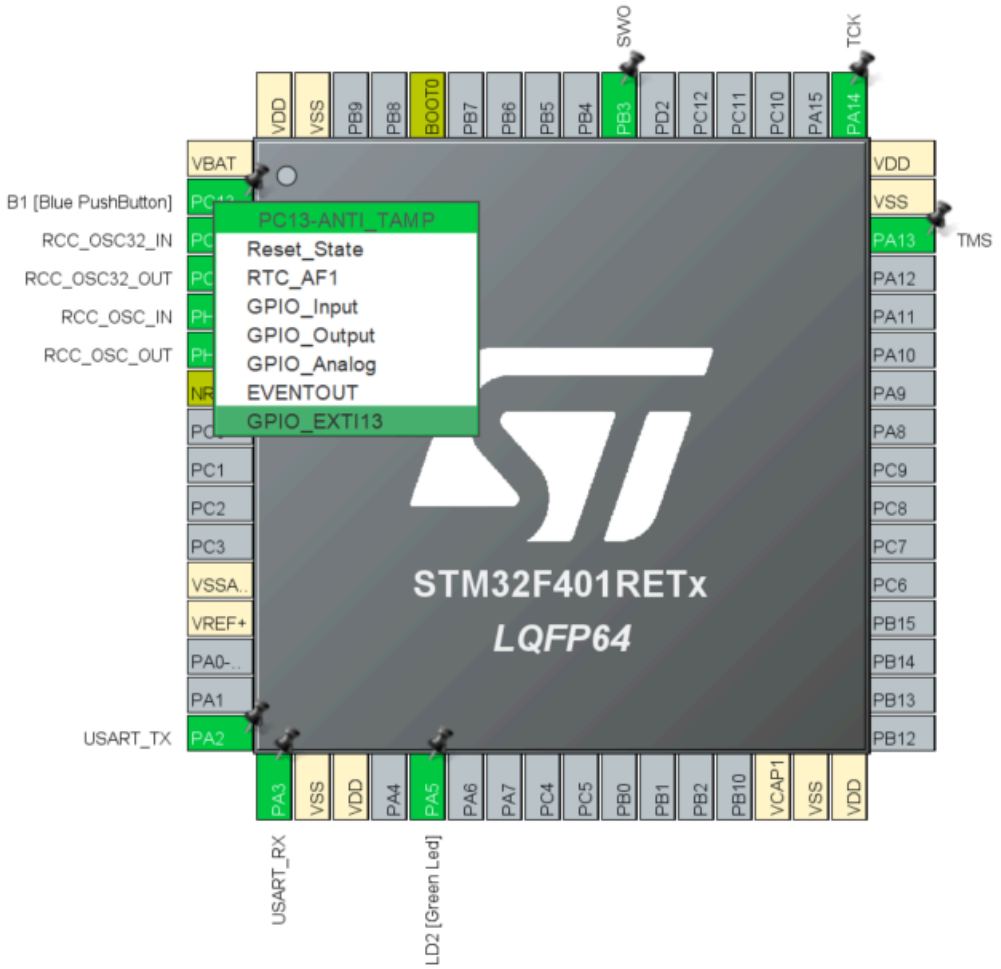
Setting Up the Project

Open STM32CubeIDE, select the previously created workspace and create a project with the name*UserButton_interrupt*.If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

User Button Pin Configuration

Open the Pinout & Configuration Tool and locate the pin associated with the user button on the boardNucleoF401RE,The User Button is connected to the PC13 pin.

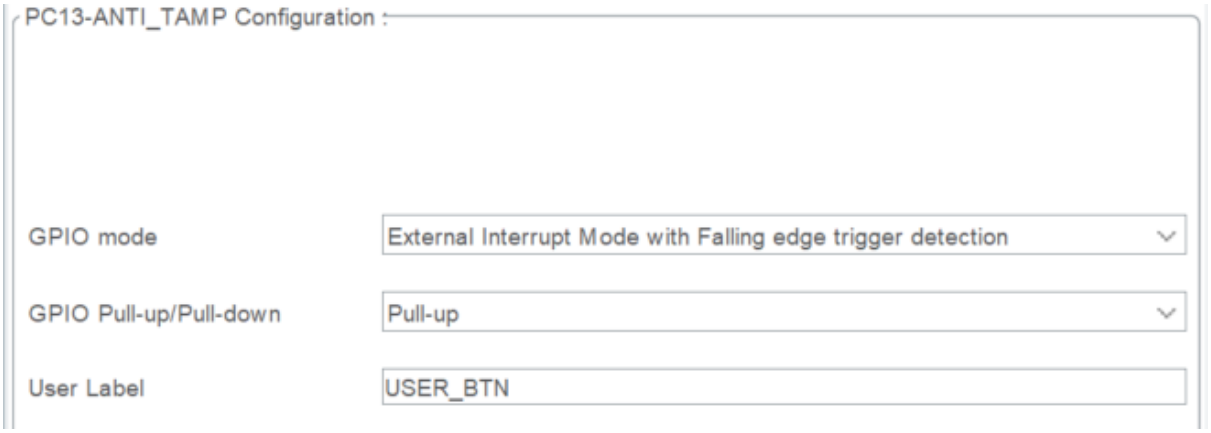
Click on the PC13 pin on the microcontroller map and set it as GPIO_EXTI13.The pin may already have been configured by Cube.



If you want to turn on the LED, you also need to configure the corresponding pin as seen previously, assigning the LED pin the label*GREEN_LED*.

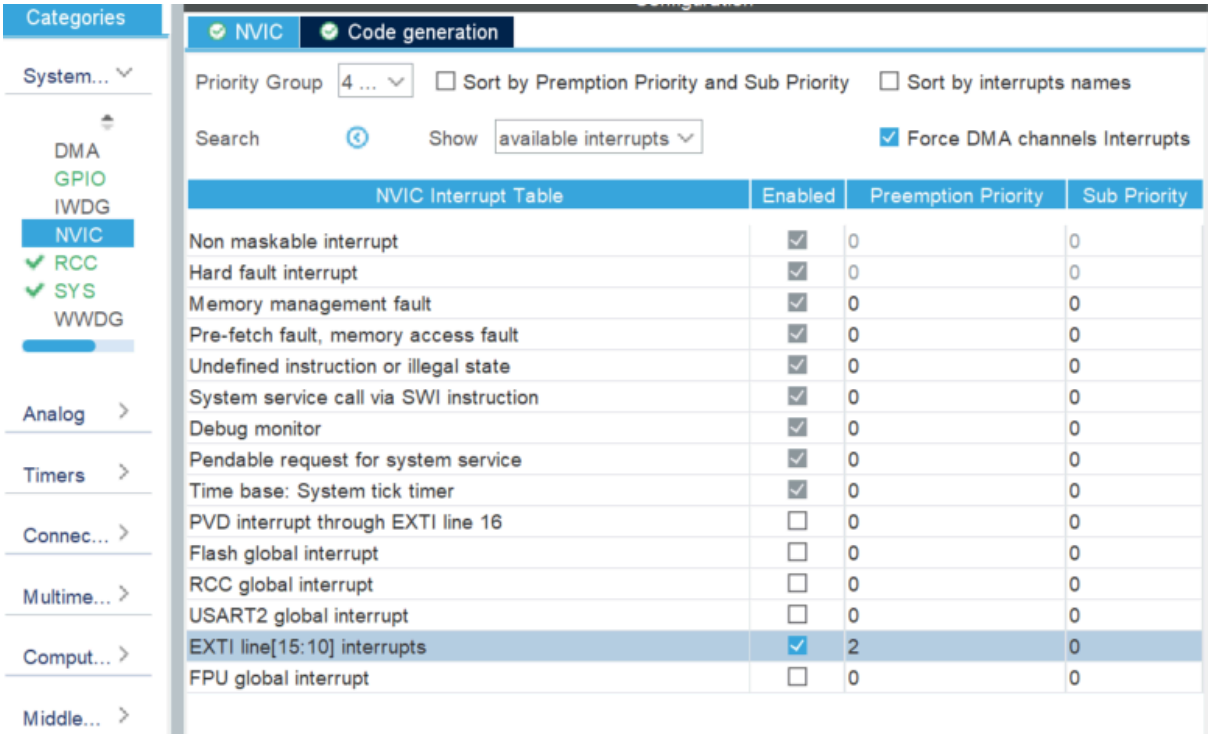
In the card **GPIO Configuration**, select the pin **PC13** and configure the following parameters:

- **Mode:** `External Interrupt Mode with Falling Edge trigger detection`
- **Pull-up/Pull-down:** `Pull-up` (optional, but often recommended to improve button stability, as the button connects the pin to GND when pressed).
- **Label:**You can optionally assign a label as `USER_BTN`.



At this point all that remains is to enable the interrupt.

- In the card **NVIC Settings** (in GPIO configuration or main configuration tab).
- Enable interrupt for **EXIT Line [15:10]** and set the desired priority.



The **priority** and the **sub-priority** interrupts are two levels of configuration offered by the NVIC (Nested Vectored Interrupt Controller) in STM32 microcontrollers. These allow you to define which interrupts should be handled first in case of conflicts or simultaneous events. For this project we can consider priorities*2*for the interrupt generated by the button.

Let's save the project and generate the code.

Button Management

The initialization function contains the pin definition *USER_BTN* and the qualification of the ’interrupt.

```
GPIO_InitStruct.Pin = USER_BTN_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(USER_BTN_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : LD2_Pin */
GPIO_InitStruct.Pin = LD2_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI15_10_IRQn, 2, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

The function remains to be implemented by callback, which is automatically invoked when an interrupt occurs.

The function to define in the file`main.c` And `HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)`. This function is designed to handle the event associated with the generated interrupt. In our project, the goal is to change the state of the green LED every time the button is pressed. To do this, we will use the function toggle on the GPIO pin associated with the green LED, inverting it state at each callback call.

```
/* Private user code
---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
   if (GPIO_Pin == GPIO_PIN_13) {
       // It changes the GREEN led status
       HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
   }
}
/* USER CODE END 0 */
```

Fill out and download the program on the Nucleo Board

Deep Dive: Interrupt Priority - Basic Concepts

When an interrupt occurs, the microcontroller must decide which ISR (Interrupt Service Routine) to execute. This is determined based on the priority and sub-priority configured in the NVIC.

The primary priority determines which interrupt takes precedence over others. For example:

- An interrupt with priority 1 takes precedence over one with priority 2.
- If two interrupts with different priorities occur simultaneously, the microcontroller executes the one with the higher priority (lower number) first.

Sub-priority is used to manage the execution order of interrupts that have the same main priority.In case of simultaneous interrupts with equal main priority, the microcontroller uses the sub-priority to decide which ISR to execute first.

In STM32, the NVIC system supports up to 16 main priority levels (depending on the device) and multiple sub-priority levels.

The number of main and sub-priority levels is determined by thePriority Groupconfigured. This configures the number of bits assigned for priority and sub-priority.

Priority Grouping divides the bits available for priority into two fields:

- **Bits per Main Priority**: Determine the main priority level.
- **Bits per Sub-Priority**: Determine the relative priority between interrupts in the same main group.

The assignment is configured with the function:

```
HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
```

The available groups are:

Priority Group	Bits per Main Priority	Bits per Sub-Priority
NVIC_PRIORITYGROUP_0	0	All bits for sub-priority
NVIC_PRIORITYGROUP_1	1	3
NVIC_PRIORITYGROUP_2	2	2
NVIC_PRIORITYGROUP_3	3	1
NVIC_PRIORITYGROUP_4	4	0 (main priority only)

If the priority group is not defined, the typical default value isGROUP_2 which assigns two bits for the priority and two for the sub-priority. This evidently means that 2 will be possible 2^2=4 *priority* and 2^2 = 4 *sub-priority*.

Chapter 7 Analog input - ADC

In this project, we will learn how to configure a GPIO pin to read an analog signal. We will use a potentiometer to turn the green LED on and off depending on whether the value read exceeds a predefined threshold or not.

Project Objectives

- Understand how the Analog-to-Digital Converter (ADC) works on a microcontroller.
- Set up a basic project in CubeIDE for reading analog signals.
- Use a potentiometer to control the value of an analog signal and implement decision logic to turn an LED on or off based on a predefined threshold.
- Configuring the UART port for sending messages

Basic Theory: Reading an Analog Signal

An analog signal is a continuous signal that can take on an infinite number of values within a range. To process it, the microcontroller uses an Analog-to-Digital Converter (ADC), which transforms the analog signal into a digital value that can be interpreted by the code. By configuring the ADC module appropriately, it is possible to read the value generated by a potentiometer connected to an analog pin and use this value to manage specific behaviors, such as turning on or off an LED.

What is an ADC (Analog-to-Digital Converter)?

An ADC (Analog to Digital Converter) is an electronic component that transforms analog signals (continuous values) into digital signals (discrete values) that can be interpreted by a microcontroller. In the case of the STM32F401RE, the microcontroller is equipped with a 12-bit ADC, which allows the analog value to be represented in a range from 0 to 4095.

The main features of the ADC in the STM32F401RE are:

- **Resolution**: 12 bit (can also be configured to 6, 8 or 10 bit to reduce conversion times).

- **Channels**: Up to 16 channels, allowing you to acquire signals from multiple analog inputs.
- **Conversion Mode**:
    - Single Conversion Mode.
    - Continuous Conversion Mode.
    - Regular and/or injected mode conversion.
- **Timer trigger**: Ability to synchronize conversions with an internal timer.
- **DMA support**: Data transfer can be automated via DMA (Direct Memory Access).

What is an ADC channel?

An ADC channel is a physical or logical input associated with a microcontroller pin. Each channel corresponds to a specific analog source that the ADC can convert. The channels available on the STM32F401RE include:

- **Analogue inputs**: Associated with GPIO pins configured in analog mode (for example, PA0 is associated with the ADC1_IN0 channel).
- **Internal channels**: Such as internal temperature sensor, VREFINT (internal voltage reference) and VBAT (battery supply voltage).

Below is the channel map of the ADC1 of the F401RE microcontroller

ADC Channel	Associated GPIO pin
ADC1_IN0	PA0
ADC1_IN1	PA1
ADC1_IN2	PA2
ADC1_IN3	PA3
ADC1_IN4	PA4
ADC1_IN5	PA5
ADC1_IN6	PA6
ADC1_IN7	PA7
ADC1_IN8	PB0
ADC1_IN9	PB1
ADC1_IN10	PC0
ADC1_IN11	PC1

ADC1_IN12	PC2
ADC1_IN13	PC3
ADC1_IN14	PC4
ADC1_IN15	PC5

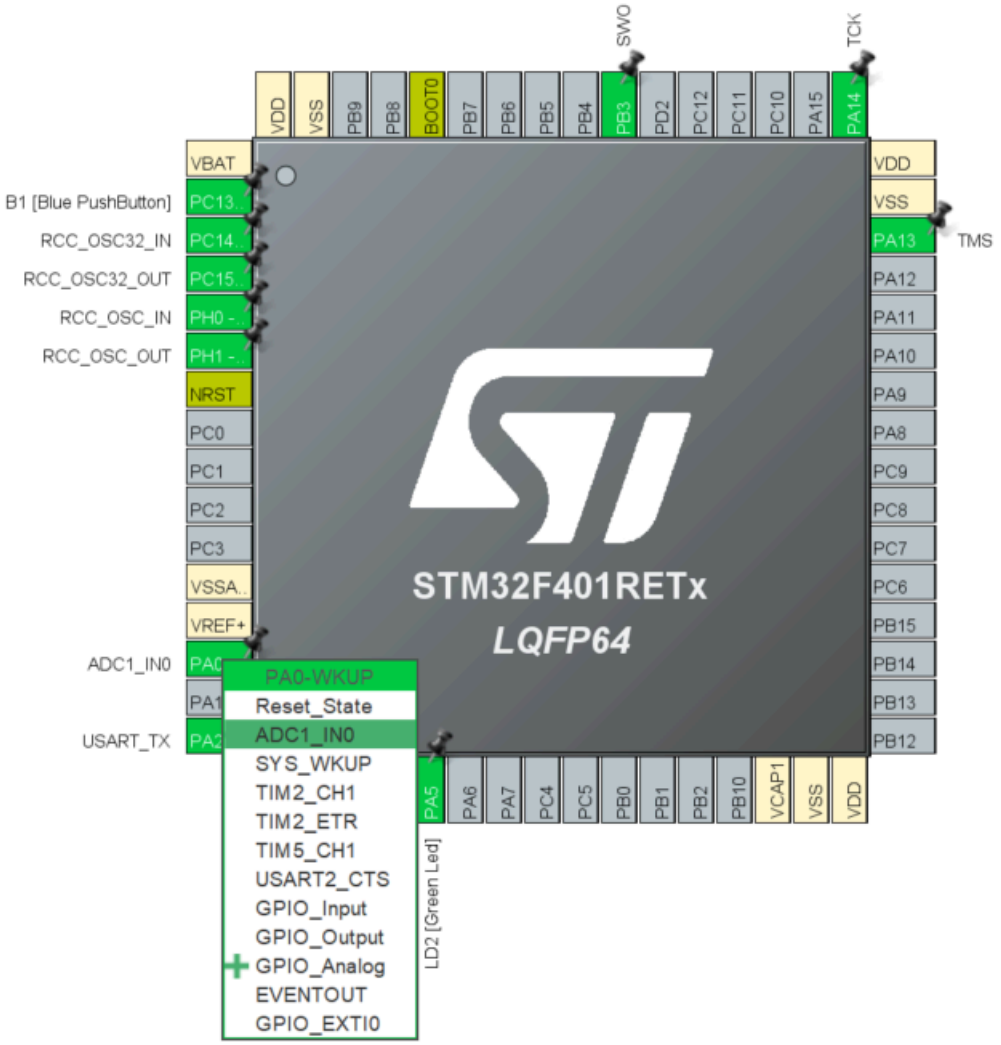
Setting Up the Project

Open STM32CubeIDE, select the previously created workspace and create a project with the name *Analog_input*. If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

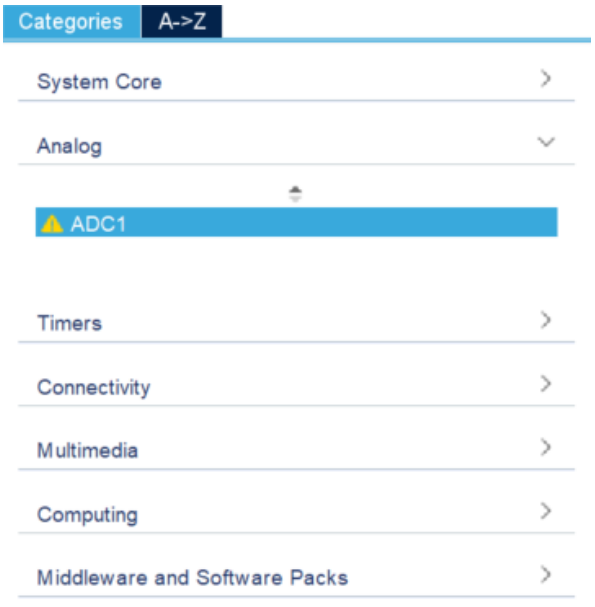
A0 pin configuration

Open the Pinout & Configuration Tool and locate the PA0 pin.

Click on the PA0 pin on the microcontroller map and set it as ADC1_IN0.

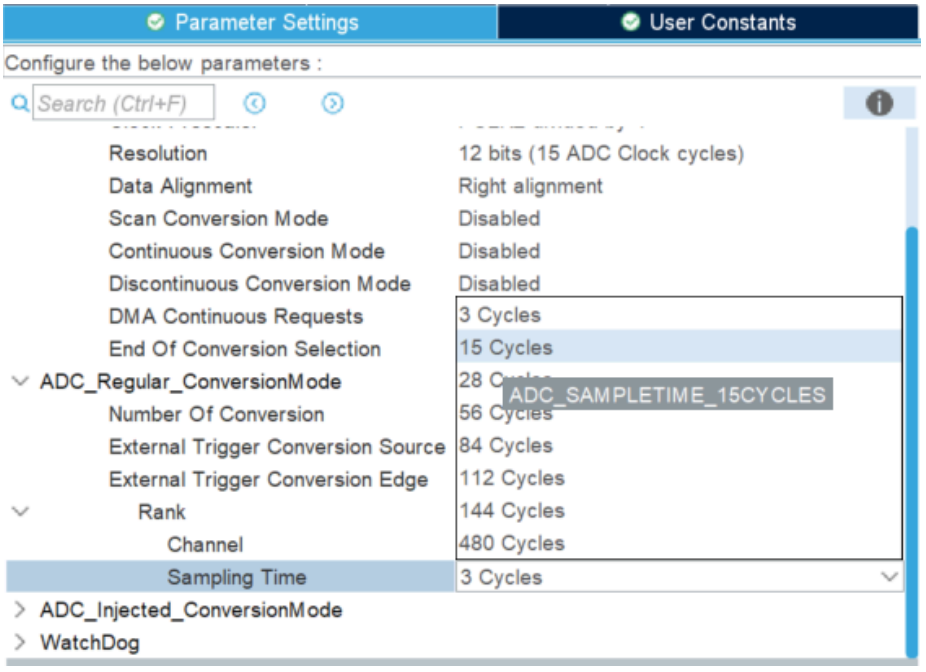


Select ADC1, the only one available for the pin *PA0*.



in field *Mode* make sure that *IN0* be selected. We need to select *IN0* and not the other channels, because the channel *0* it is the one associated with the pin *PA0*

In *Parameters Settings* we leave everything unchanged except the parameter *Sampling Time* since if you want to use all 12 bits of the converter, 15 cycles will be necessary



The *UART2* and the *LED* green are already initialized, having requested initialization to the values of default during project creation. If necessary, refer to the chapters dedicated to their initialization.
To verify that the LED and the UART2 pin are initialized, simply check that the respective pins are colored green.

We are ready to generate the code. We do this by saving the project, in which caseCubeIDEit will ask us if we want to generate it, or asking explicitly in the menu *Project/Generate Code*.

If we want to print the value read by the ADC on the screen, we will use functions contained in the libraries stdio *string*. We must therefore proceed to include the respective header files in the include section provided for the *USER CODE*.

```
/* Private includes
----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <string.h>
#include <stdio.h>
/* USER CODE END Includes */
```

To use the value read by the ADC, we need two private variables. One of type*uint16_t* to store the read value, encoded in 12 bits, and an array of characters for the message to send. We insert them into the section allocated to *Private*

```
/* USER CODE BEGIN PV */
uint16_t measure = 0;
char msg[20];
/* USER CODE END PV */
```

To read the ADC we need to add the following code in the main loop section.

```
while (1)
 {
   /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
       HAL_ADC_Start(&hadc1);
       HAL_ADC_PollForConversion(&hadc1, 20);
       measure = HAL_ADC_GetValue(&hadc1);
       sprintf(msg, "Value read: %hu\r\n", measure);
       HAL_UART_Transmit(&huart2 , (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);
       HAL_Delay(50);
 }
```

Start the analog-to-digital converter (ADC) with HAL_ADC_Start, so as to start converting the analog signal into a digital value.

Wait for the conversion to complete using HAL_ADC_PollForConversion, with a 20 millisecond timeout to avoid infinite locks.

Retrieve the converted value via HAL_ADC_GetValue and stores it in the "measure" variable.

Formats the read value into a string using sprintf, preparing it for broadcast.

Transmits the formatted message to the UART port(HAL_UART_Transmit) with a maximum timeout, ensuring that the message is sent successfully.

Introduces a short pause of 50 milliseconds (HAL_Delay) before performing any further operations.

Fill out and download the program on the Nucleo board.

To view messages sent by the Nucleo card you will need to use a client, open the connection on the port assigned to the Nucleo by the operating system, selecting 115200 bps as the connection speed.

Chapter 8 Timers

In this chapter we will delve into the use of timers on the STM32 microcontroller, illustrating the fundamental concepts and guiding you through the practical configuration of the project.

Project Objectives

The main goal of this project is to make you experiment and understand the operation of timers in the STM32 environment. In particular, you will learn to:

- **Generate periodic events:** Set up the timer to perform actions at regular intervals.
- **Handling interrupts:** Configure timer-derived interrupts to optimize task handling without overloading the processor.
- **Experiment in a practical way:** Put theoretical knowledge into practice with real-world exercises that include timer configuration using tools such as STM32CubeIDEand the use of the HAL.

Basic Theory: What is a Timer?

Timers are essential hardware devices for time control within embedded systems, as they allow you to count clock pulses and measure time intervals precisely.

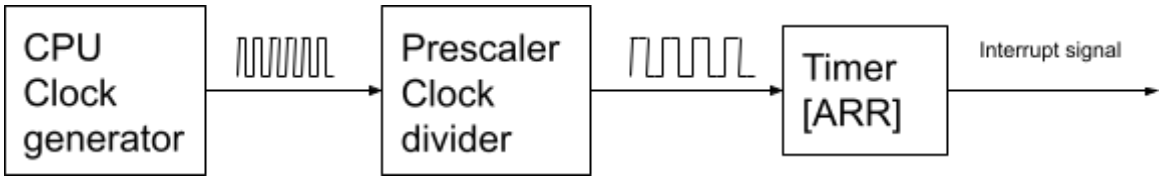
With a timer we can both measure time intervals and generate timed events, using the counting of pulses of a clock signal with a known frequency. Knowing the frequency of the clock and counting the number of pulses, it is possible to calculate the elapsed time interval precisely.

Using a timer correctly involves knowing how to balance the clock frequency, the prescaler and the autoreload value to obtain the desired timing resolution and ensure efficient event management, which is essential for the correct functioning of embedded applications.

To trigger an event after a certain period of time, you set a comparison value, which corresponds to the number of pulses to be counted, and you can adjust the counting interval using a prescaler.

The prescaler reduces the input frequency by dividing the clock by a predefined factor, thus allowing for longer time intervals. In addition, the timer has a register called ARR (Auto-Reload Register) in which the maximum value of the count:once this value is reached, the counter resets and, if properly configured, generates an event (for example, the

activation of an interrupt). In this way, the combination of prescaler and ARR allows to obtain the desired time resolution and to efficiently manage time-based operations.



In addition, timers operate in multiple modes, including PWM signal generation, input capture to measure the duration of external events, and compare output to handle time-bound events, providing flexibility that allows the microcontroller to perform timing operations independently and reactively. This topic will be covered in the next chapter.

Setting Up the Project

OpenSTM32CubeIDE,select the previously created workspace and create a project with the name*Timer*.If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

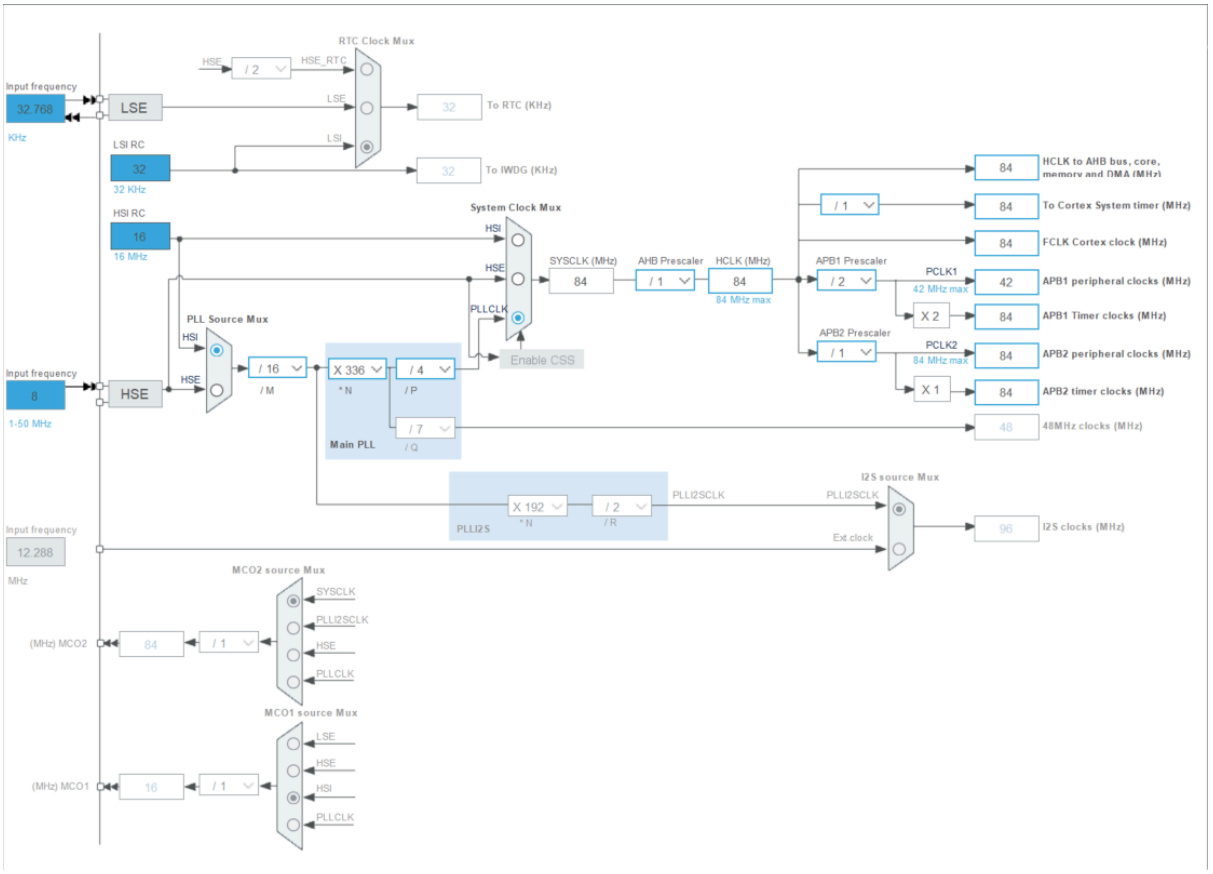
By double clicking on the file*Timer.ioc*let's access the microcontroller configuration.

From the page*Pinout & Configuration* let's move on to the page*Clock Configuration*. On this page you can make some configuration changes to the peripheral clock.

We can change some of the blocks of the *tree*, that is, those relating to the prescalers, AHB and APB1 and APB2.

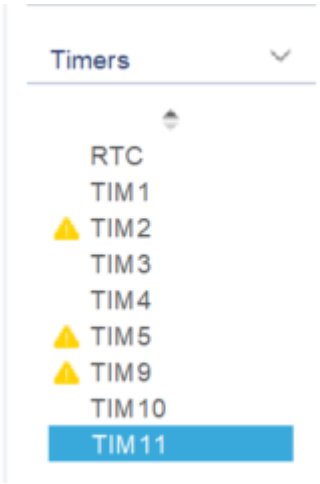
The AHB prescaler is a frequency divider applied to the main clock of the microcontroller to obtain the operating frequency.dell'AdvancedHigh-performance Bus (AHB). This bus connects the processor core to high-speed peripherals, and through the prescaler, the system clock is reduced to meet the operating specifications of different parts of the device, improving energy efficiency and ensuring system stability.

Advanced Peripheral Buses (APBs) are internal buses that connect the microcontroller core to peripherals, playing a key role in the architecture of ARM-based embedded systems, such as the STM32. In particular, these buses are designed to handle communication with devices that do not require high data transfer rates, such as UART, I2C, SPI, timers, and ADCs. In STM32 microcontrollers, APBs are generally divided into two groups: APB1, intended for lower-frequency peripherals, and APB2, which supports devices that operate at higher frequencies. The frequency of each bus is obtained by applying a prescaler to the main clock (derived from AHB),in order to adapt the operating speed to the specific needs of the connected peripherals, thus ensuring efficient communication and optimising energy consumption.

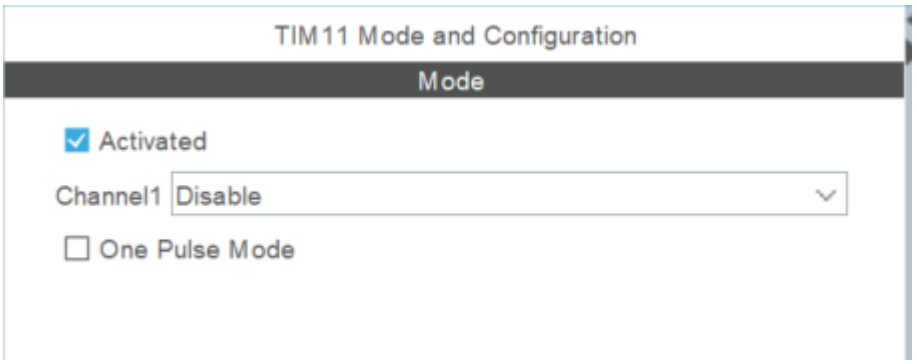


For the moment, do not modify it, and we go back to the page of *Pinout & Configuration*.

We select*Timers*



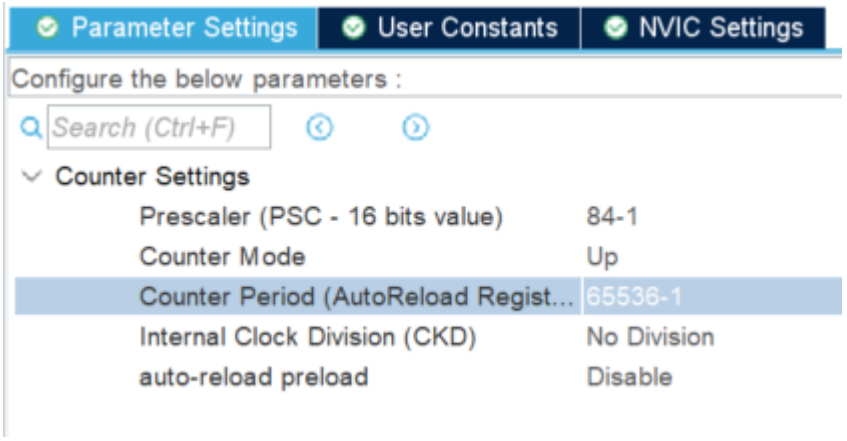
and then *TIM11*and we enter the timer configuration mode



Let's clear the field*Activated*to enable the timer and leave the other fields at their default values.

In the Parameter Settings we define the value of the prescaler. Since the value starts from 0 and corresponds to a prescaler of 1, it can be convenient to insert an expression in which 1 is subtracted from the desired value. So, wanting a prescaler of 84, and having to insert 83, we insert the expression 84-1.

Let's do the same thing for valuedell'ARR. If we want to count the maximum value, that is 65536, but have to insert 65535, we will write the expression 65536-1.



Since the clock frequency is 84 MHz, and having selected a prescaler value of 83 (84-1), the timer will see a signal with a frequency equal to:

$freq_{desired} = 84Mz \div 84 = 1MHz$

i.e. 1 tick every$1 \div 1MHz = 1us$

Having entered ARR = 65536-1 we will be able to count at most 65536 us or 65.536 ms

In general, the following relationships apply:

$$PSC = \frac{TIM\ CLK}{Desired\ frequency} - 1$$

$$ARR = \frac{Desired\ frequency}{Timer} - 1$$

As for the*Counter Mode*, the value*Up*set the timer for an increasing count. finally we disable the *auto-reload preload*.

We save the project and accept the code generation.

CubeIDEwill create an initialization function with the parameters entered in the file*Timer.ioc*

```
/**
 * @brief TIM11 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM11_Init(void)
{
 /* USER CODE BEGIN TIM11_Init 0 */
 /* USER CODE END TIM11_Init 0 */
 /* USER CODE BEGIN TIM11_Init 1 */
 /* USER CODE END TIM11_Init 1 */
 htim11.Instance = TIM11;
 htim11.Init.Prescaler = 84-1;
 htim11.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim11.Init.Period = 65536-1;
 htim11.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim11.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
 if (HAL_TIM_Base_Init(&htim11) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN TIM11_Init 2 */
 /* USER CODE END TIM11_Init 2 */
```

If we want to send the elapsed time via serial port, we include the standard IO (stdio.h) library

We do it in the section *user* dedicated to includes.

```
/* Includes ------------------*/
#include "main.h"
/* Private includes ---------*/
/* USER CODE BEGIN Includes  */
#include <stdio.h>
/* USER CODE END Includes    */
```

Then you need to define some variables, the message buffer that we will send on the serial port, one to store the length of the messages and one the value read by the timer

```
int main(void)
{
 /* USER CODE BEGIN 1 */
     char uart_buf[50];
     int uart_buff_len;
     uint16_t timer_val;
 /* USER CODE END 1 */
```

We then send a splash message to the serial port and start the timer *TIM11*

```
/* USER CODE BEGIN 2 */
 // SPLASH MESSAGE
uart_buff_len = sprintf(uart_buf, "Timer\r\n");
HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buff_len, HAL_MAX_DELAY);
 // Starting the timer
HAL_TIM_Base_Start(&htim11);
 /* USER CODE END 2 */
```

At this point we are ready to read the timer value. We make two consecutive readings separated by a known delay of 50 ms, and after measuring the elapsed time by difference, we send it to the serial port.

Finally we wait another 1s so that we have spaced measurements.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
     timer_val = __HAL_TIM_GET_COUNTER(&htim11);
     HAL_Delay(50); //ms
     timer_val = __HAL_TIM_GET_COUNTER(&htim11) - timer_val;

     uart_buff_len = sprintf(uart_buf, "%u us\r\n", timer_val);
     HAL_UART_Transmit(&huart2, uart_buf, uart_buff_len, HAL_MAX_DELAY);

     HAL_Delay(1000); //ms

    /* USER CODE END WHILE */
```

Fill out and download the program on the Nucleo board.

To view messages sent by the Nucleo card you will need to use a client, open the connection on the port assigned to the Nucleo by the operating system, selecting 115200 bps as the connection speed.

The displayed messages will report the 50ms wait time expressed in microseconds.

Chapter 9 PWM

In this chapter we will delve into the use of PWM (Pulse-Width Modulation) signals on the STM32 microcontroller, illustrating the fundamental concepts and guiding you through the practical configuration of the project.

Project Objectives

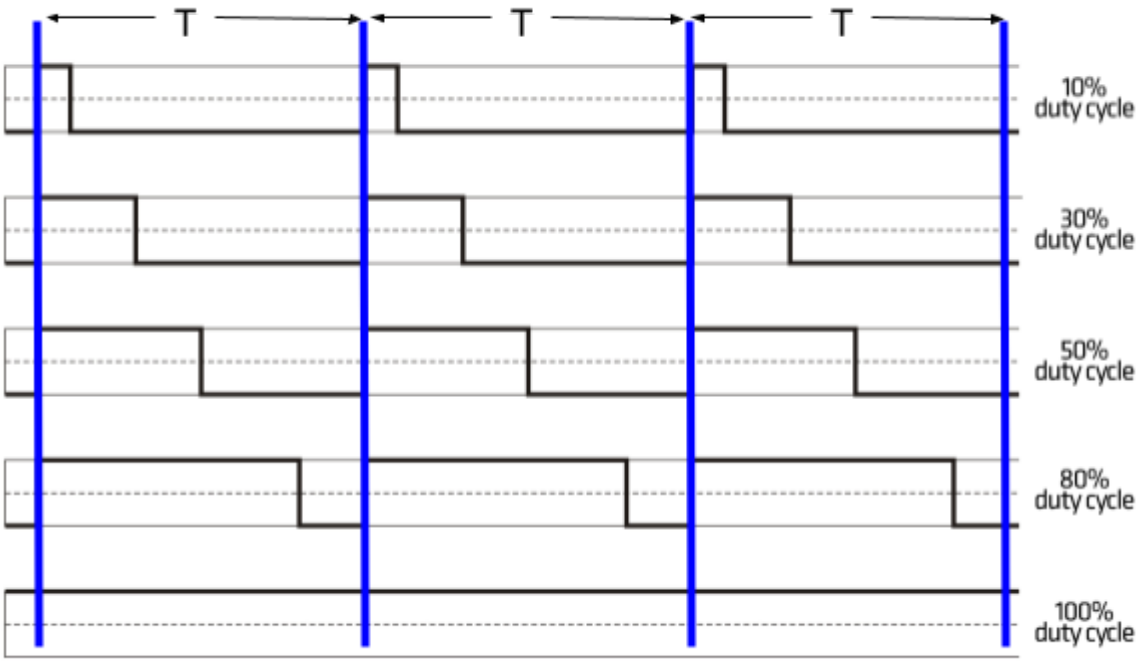
The main goal of this project is to make you experience and understand the operation of PWM in the STM32 environment. In particular, you will learn to:

- **Generate PWM signals:** Configure timer channels to produce PWM outputs with variable frequency and duty cycle.
- **Adjust the duty cycle:** Change the pulse width to control the light intensity of an LED.
- **Manage advanced modes:** Using PWM in center-aligned mode, generation of complementary outputs with dead-time and fault management.

Basic Theory: What is PWM

Pulse-width modulation (PWM) is a technique for generating a periodic digital signal in which the duration of the “high” level (duty-cycle) is variable, while keeping the overall frequency constant.

- **PWM Frequency:** It determines the period T=1/f of the signal and is defined by how the prescaler and the Auto-Reload Register (ARR) value of the timer are set.
- **Duty-Cycle:** Indicates the percentage of time the signal remains “high” within each period, and is controlled by the comparison value (CCR). A duty cycle of 50% means half the period is “high” and half is “low”.



The main registers involved in generating a PWM signal are:

- **Prescaler (PSC):** divides the base clock frequency to slow down the timer count, allowing for lower PWM frequencies or finer resolutions.
- **Auto-Reload Register (ARR):** defines the maximum value of the counter; when it is reached, the timer resets and starts again from zero, thus establishing the PWM period.
- **Capture/Compare Register (CCRx):** contains the threshold value for the duty cycle; when the counter exceeds this value, the output changes state (from “high” to “low” or vice versa).

and therefore to generate a PWM signal we must act appropriately on these registers.

The first step is to set up the registers **PSC** and **ARR** in order to obtain the desired frequency:

$$f_{PWM} = \frac{f_{CLK}}{(PSC + 1)(ARR+1)}$$

Once the frequency is set, we need to define the duty cycle and we do this by setting the **CCR** to define when the signal should return to zero.

$$Duty\ Cycle = \frac{CCR}{ARR + 1} \times 100\%$$

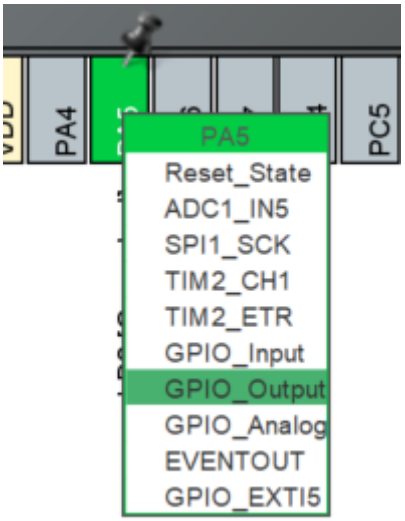
Setting Up the Project

Open STM32CubeIDE, select the previously created workspace and create a project with the name*PWM*.If you don't remember how to do it, you can review the previous chapter, the procedure for creating and configuring a project does not change.

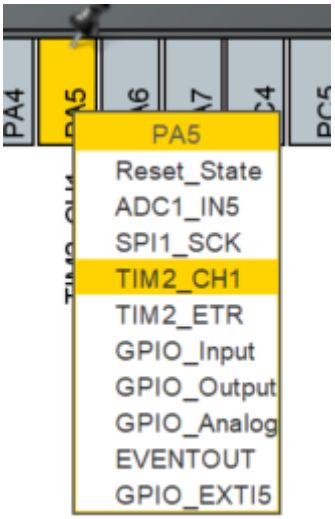
By double clicking on the file *pwm.ioc* let's access the microcontroller configuration.

A5 pin configuration

The LD2 led is connected to pin A5, so we need to configure the timer that this pin refers to. We select pin A5 in the configurator, which if we choose to proceed with the default configuration during the project creation phase, will be initialized asGPIO_Output.

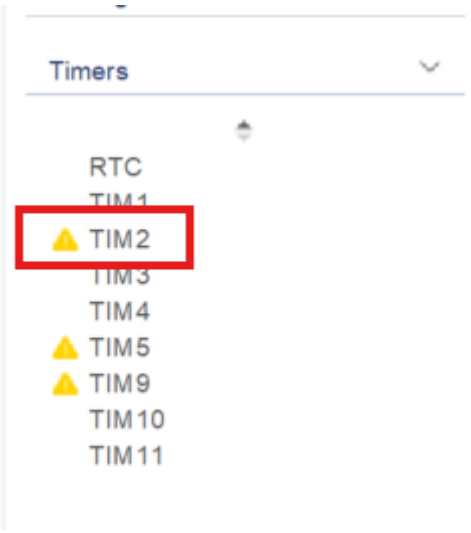


We select channel 1 of timer 2 for this pin

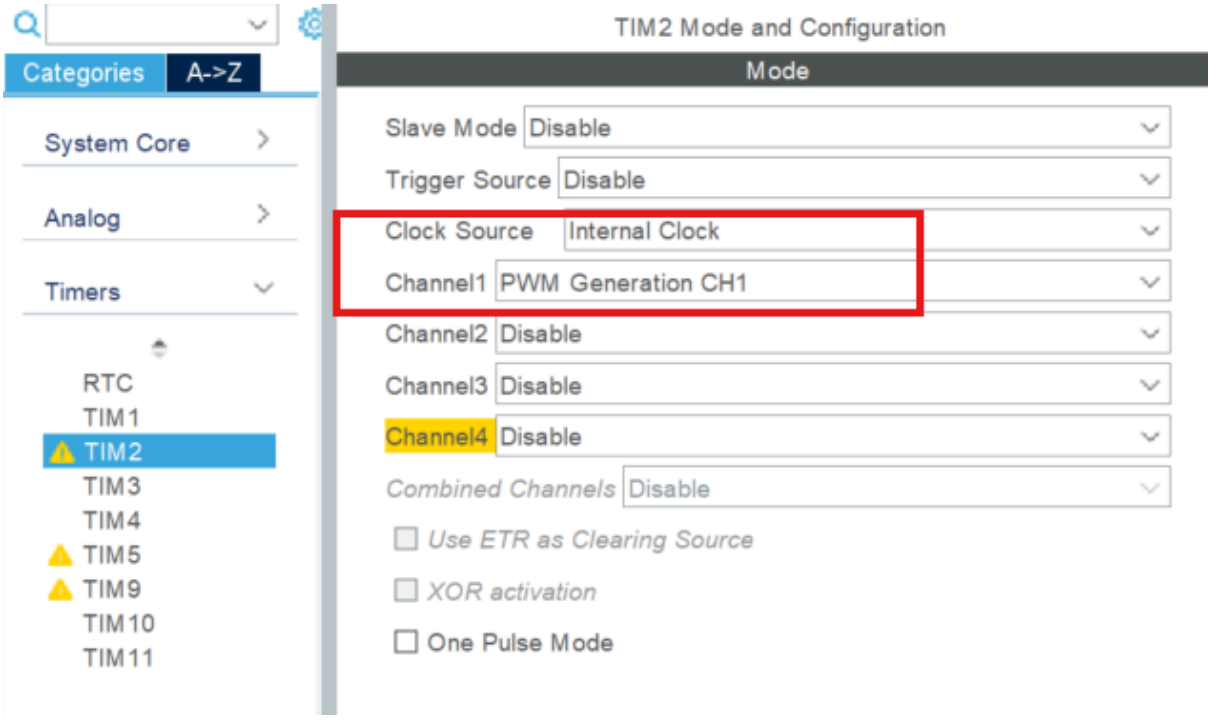


The selection turns yellow because it is necessary to proceed with other configuration steps at the end of which, if everything has been configured correctly, it will turn green.

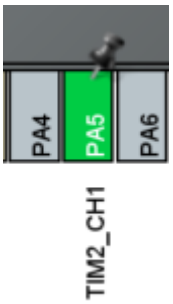
In the menu *Timers* we select the TIM2



In the configuration panel *TIM2 Mode and Configuration* we select for channel 1 the generation of a PWM signal and as *Clock Source* the Internal Clock.



This configuration is enough to make the PA5 pin green, but the signal definition *pwm* desired, it is not yet complete.

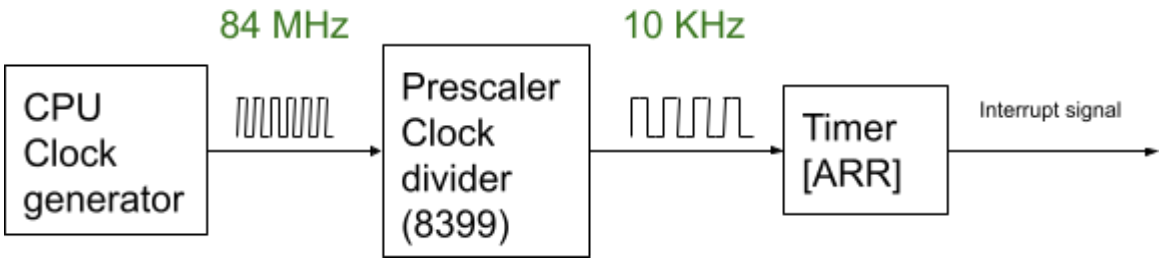


Example 1

As a first case, we want to turn on the Nucleo board LED for 1 second and turn it off for 1 second.

To achieve this, we need a signal that has a period of 2 seconds with a duty cycle of 50%.

For the*Prescaler* we enter the value 8399 to bring the clock signal (84 MHz) to 10 KHz.



PSC+1 = 8400; lowers 84 MHz to 10 kHz counting. Remember that the prescaler is a register whose value starts from*zero*Entering the value 8399 is equivalent to entering a scaling factor 8399+1=8400.
So at the output of the prescaler we will have a clock signal of frequency:

$$f_{prescaler} = \frac{84 \times 10^6}{8400} = 10^4 = 10\ KHz$$

The signal obtained from the prescaler is therefore a signal that has a period of:

$$T_{prescaler} = \frac{1}{10^4} = 0,0001\ s$$

To calculate the value to be inserted into the register*ARR*, that is, the duration of the period of 2 seconds, we need to understand how many pulses of the signal there are*10 KHz*we have to count.

We want the signal to be high for 1 second and low for the remaining second of the period.

We then need to count a number of pulses of the 10KHz signal to make 2 seconds.

$$(ARR + 1) \times T_{prescaler} = 2\,s$$

and therefore

$$ARR = \frac{2}{T_{prescaler}} - 1 = 20000 - 1 = 19999$$

All that remains is to define the *Ton* that is, the duration of the pulse within the period. We do this by inserting the value *9999* in the field*Pulse*.

Indeed

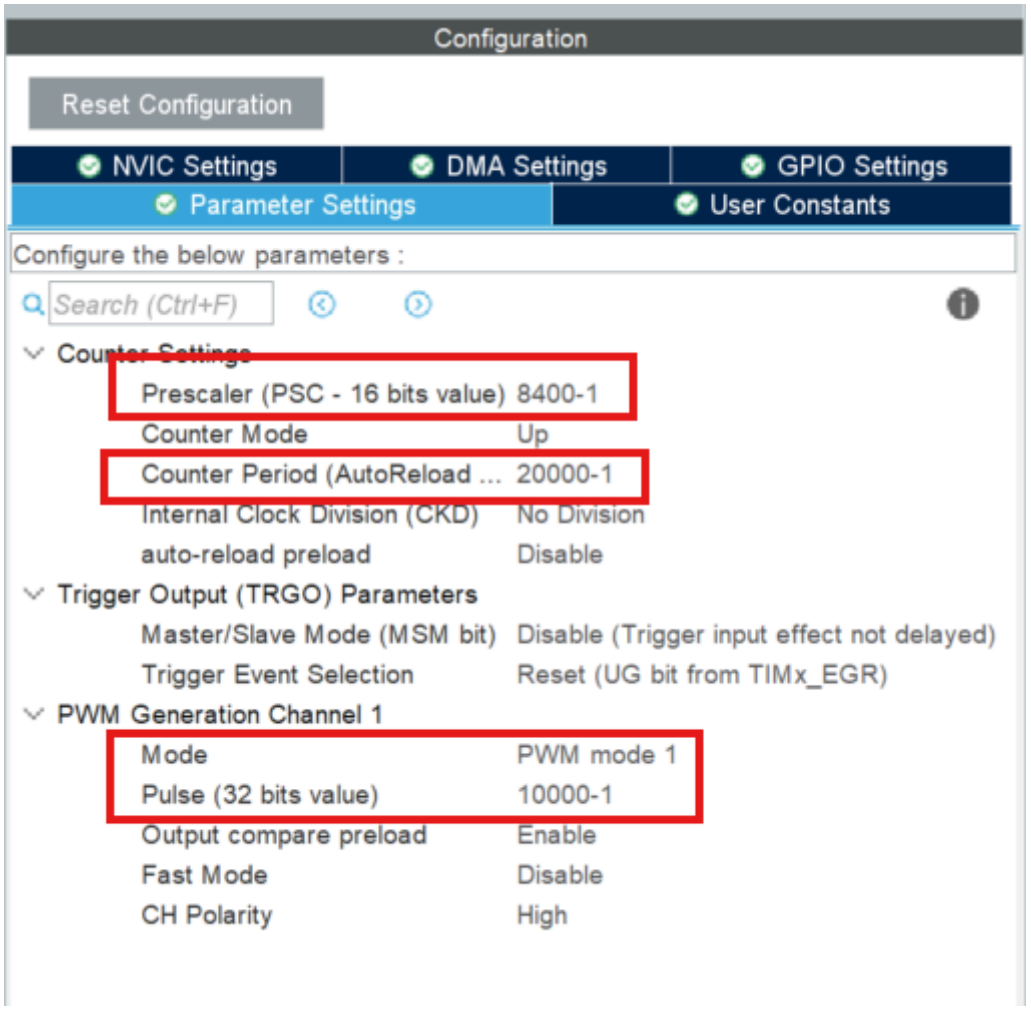
$$Duty\ Cycle = \frac{CCR + 1}{ARR + 1} = \times 100\%$$

from which:

$$CCR = \frac{Duty\ Cycke \times (ARR+1)}{100\%} - 1 = \frac{50\% \times (ARR+1)}{100\%} - 1 = \frac{ARR+1}{2} - 1 = \frac{20000}{2} - 1 = 10000 - 1$$

In CubeIDE, in the panel*Parameter settings*, the value assigned to the CCR register is indicated with*Pulse*.

The following figure shows the panel*Parameter Settings* with the values to be assigned, leaving the others at their default value.



To improve readability, it is good practice to enter the desired value in the registers subtracting one to take into account that the values start at zero.

At this point we can generate the code by saving the project and accepting the code generation.

All that remains is to add the code to start generating the signal.

```
/* USER CODE BEGIN 2 */
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
 /* USER CODE END 2 */
```

Compile and download into Nucleo as previously done. We will see the green LED flashing.

Example 2:

Now we want to get an effect *fade-in* on the LED, that is, we want it to start from off and gradually light up and then start the cycle again.

To get the effect *fade-in*, we need to get the file back*pwm.ioc*and make changes.

We want to bring the PWM frequency to 1 KHz starting from a frequency downstream of the prescaler of 1 MHz

In the *prescaler* we insert the value *84-1*. Indeed:

$$f_{prescaler} = \frac{f_{clock}}{PSC + 1}$$

Therefore

$$PSC = \frac{f_{clock}}{f_{prescaler}} - 1 = \frac{84 \times 10^6}{10^6} - 1 = 84 - 1 = 83$$

As for the value to be entered in the*ARR*, let's start from the frequency of 1 KHz that we want the PWM signal to have.

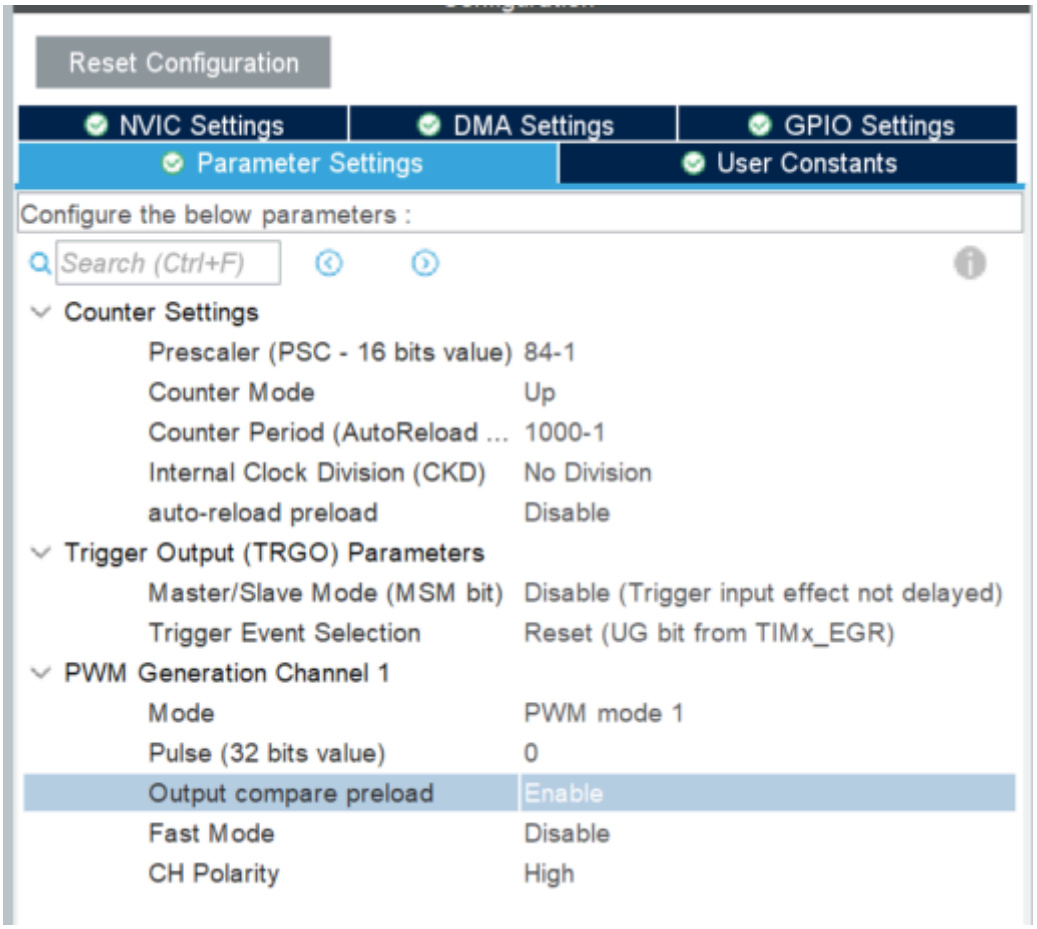
$$T_{pwm} = \frac{1}{f_{pwm}} = \frac{1}{10^3} s = 10^{-3} s = 0,001\, s = 1ms$$

$$(ARR + 1) \times T_{prescaler} = 1ms$$

$$(ARR + 1) \times 10^{-6} = 0,001$$

$$ARR = \frac{0.001}{10^{-6}} - 1 = 10^3 - 1 = 999$$

So the value to be inserted in the*ARR*it's 999.
As for the*CRR (Pulse)* we leave it to *0* and we will modify it appropriately from code.




```
/* USER CODE BEGIN 2 */
 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);  // avvia subito il PWM
 uint32_t maxPulse = htim2.Init.Period;  // = 999
 /* USER CODE END 2 */
  /* USER CODE BEGIN WHILE */
 while (1)
 {
        for (uint32_t pulse = 0; pulse <= maxPulse; pulse++)
          {
            __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, pulse);
            HAL_Delay(1);  // 1 ms per step → fade di ~1 s
          }
        /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
 }
 /* USER CODE END 3 */
}
```

The code starts the signal generation *pwm*, and reads the period value from the register *ARR* to assign it to the variable *max Pulse.* The cycle *for* assign to the *ton,* that is to *pulse,* a value that starts from *0* and reaches the duration of the period, that is, 100% of the *dutycycle*.

At this point we can generate the code, compile and download to the Nucleo as we did before. We will see the brightness of the green LED increase from zero to its maximum value.