

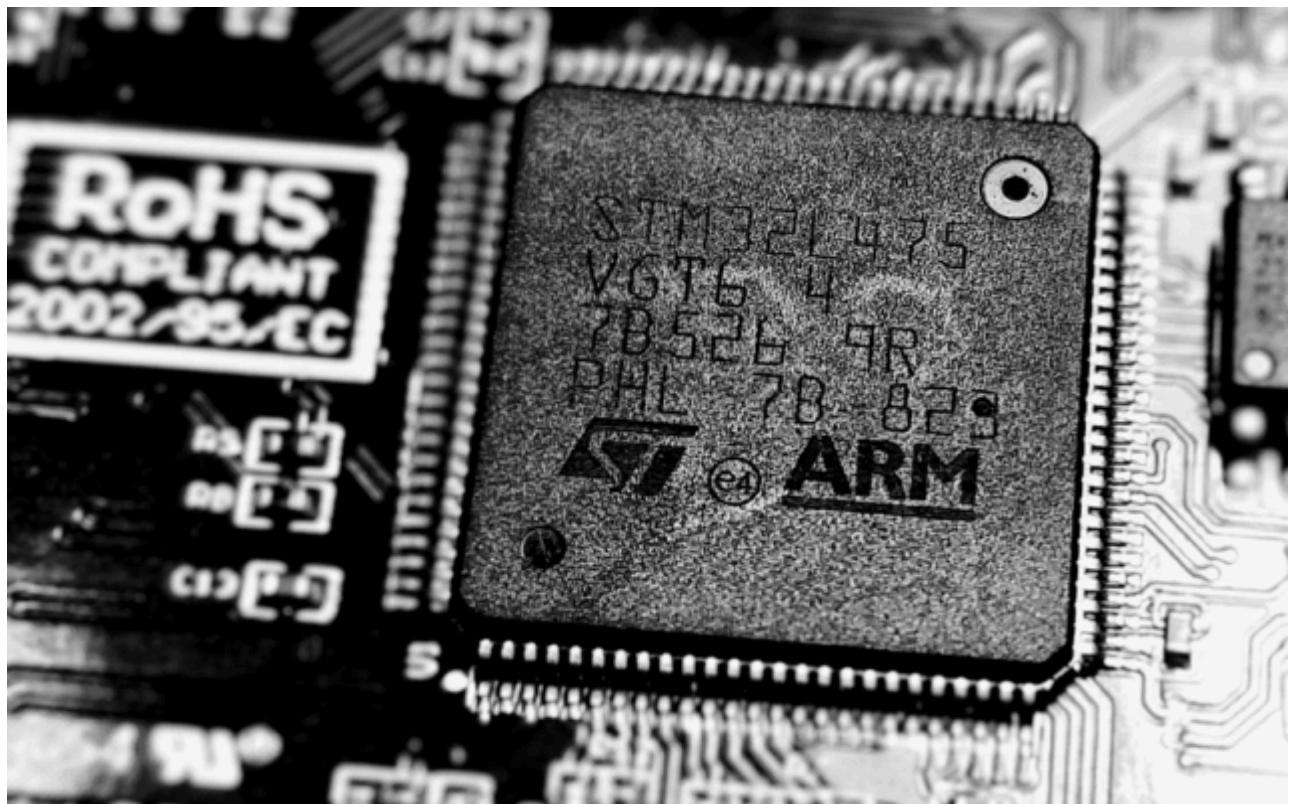


# Il Cuore dei Sistemi Embedded

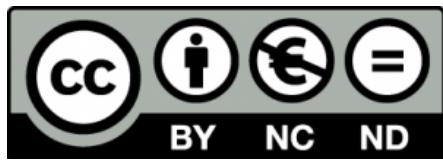
## Getting Started

Teoria, Codice e Applicazioni con Microcontrollori STM32

Mauro D'Angelo



The Heart of Embedded Systems © 2025 by Mauro D'Angelo is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-nd/4.0/>



## Indice

<b>Introduzione</b>	7
<b>Capitolo 1 Microcontrollori</b>	9
Introduzione ai Microcontrollori	9
Il numero di bit della CPU	10
Cosa indirizza un microcontrollore?	11
Come funziona l'indirizzamento	11
La mappa di memoria	12
Applicazioni	14
Alternate Functions nei Microcontrollori	15
Cosa Sono le Alternate Functions	15
Perché sono importanti	15
Come Funzionano	15
Alternate functions del microcontrollore STM32F401RE	16
<b>Physical Computing</b>	20
<b>Sensori</b>	21
Esempio: Sensore di Temperatura sulla Scheda di Espansione IKS4A1	22
<b>Gli attuatori</b>	23
Attuatori Elettrici	23
Attuatori Idraulici	25
Attuatori Pneumatici	25
Attuatori Termici	26
<b>I segnali digitali</b>	27
<b>I segnali analogici</b>	28
<b>Conversione Analogico/Digitale</b>	29
Esempio pratico: ADC su STM32F401RE	30
Le principali applicazioni dell'ADC	30
<b>I protocolli di comunicazione</b>	31
SPI - Serial Peripheral Interface	31
I2C	32
Serial	34
<b>Ricapitoliamo</b>	37
<b>Capitolo 2 La scheda Nucleo F401RE</b>	38
STM32 Microcontroller Naming Convention	39
Funzionalità hardware della NucleoF401RE	39
Etichette dei pin	40
CubelDE	43
Installazione	44
Requisiti di Sistema	44
Scaricare STM32CubelDE	44
Installazione su Windows	45
Installazione su macOS	45
Installazione su Linux	45

Configurazione di STM32CubeIDE	45
Introduzione a STM32CubeMX	46
<b>Capitolo 3 GPIO output</b>	<b>47</b>
Obiettivi del Progetto	47
Teoria di Base: Cos'è un GPIO?	47
Configurare il Progetto	47
Compilare il progetto	52
Programmazione del microcontrollore	54
Il file main.c	54
Configurazione del Pin	56
Lampeggiare il LED	60
<b>Capitolo 4 La comunicazione seriale</b>	<b>62</b>
Obiettivi del Progetto	62
Teoria di Base: Comunicazione Seriale Asincrona (UART)	62
Modalità di Gestione della Ricezione	63
Configurare il Progetto Serial Polling	63
Configurazione dei pin della USART2	64
Configurare il Progetto Serial Interrupt	71
Configurazione dei pin della USART2	71
<b>Capitolo 5 GPIO input - polling</b>	<b>72</b>
Obiettivi del Progetto	72
Teoria di Base: Lettura di un GPIO di input	72
Configurare il Progetto	72
Configurazione del pin User Button	73
Lettura del Pulsante	75
Approfondimento: I resistori di pull up/down	76
<b>Capitolo 6 GPIO input - interrupt</b>	<b>78</b>
Obiettivi del Progetto	78
Teoria di Base: Generazione di un interrupt	78
Configurare il Progetto	80
Configurazione del pin User Button	80
Gestione del pulsante	82
Approfondimento: Priorità degli Interrupt - Concetti di Base	83
<b>Capitolo 7 Analog input - ADC</b>	<b>85</b>
Obiettivi del Progetto	85
Teoria di Base: Lettura di un Segnale Analogico	85
Cos'è un ADC (Analog-to-Digital Converter)?	85
Cos'è un canale ADC?	86
Configurare il Progetto	87
Configurazione del pin A0	87
<b>Capitolo 8 Timers</b>	<b>92</b>
Obiettivi del Progetto	92
Teoria di base: Cos'è un Timer	92
Configurare il Progetto	93

<b>Capitolo 9 PWM</b>	<b>99</b>
Obiettivi del Progetto	99
Teoria di Base: Cos'è il PWM	99
Configurare il Progetto	101
Configurazione del pin A5	101
Esempio 1	103
Esempio 2:	105

Perlatecnica è un'associazione no profit di promozione sociale nata nel 2014 per iniziativa di un gruppo di appassionati di ingegneria e informatica, con l'obiettivo di diffondere la cultura digitale attraverso attività di formazione, ricerca e divulgazione. I suoi volontari – ingegneri, informatici e semplici appassionati – organizzano seminari, workshop e corsi online, realizzando materiali didattici e piattaforme e-learning liberamente accessibili.

Con questo libro vogliamo portare avanti la stessa missione: rendere accessibile a tutti, dagli studenti ai professionisti, la conoscenza sui microcontrollori STM32 e sulle tecnologie di base dell'elettronica e del physical computing. Attraverso esempi pratici e guide passo-passo, l'intento è quello di trasformare concetti teorici in competenze operative, stimolando la creatività e l'autonomia di chi si avvicina per la prima volta a questi argomenti.

<https://www.perlatecnica.it/>

# Introduzione

Questo libro è pensato per accompagnarti dalla teoria alle prime esperienze pratiche, con esercizi e snippet di codice pronti all'uso. L'obiettivo è che, al termine della lettura, tu possa non solo comprendere i principi di funzionamento dei microcontrollori STM32, ma anche saperli applicare in autonomia ai tuoi progetti. Un libro non potrà mai essere esaustivo dell'argomento trattato, ma un libro come questo può dare al lettore gli spunti adeguati per condurre ricerche e approfondimenti personali.

Il codice degli esempi è disponibile al seguente link:

[https://github.com/Perlatecnica/TheHeartOfEmbeddedSystems/tree/master/GettingStarted/WS\\_CoreOfEmbeddedSystems](https://github.com/Perlatecnica/TheHeartOfEmbeddedSystems/tree/master/GettingStarted/WS_CoreOfEmbeddedSystems)

Nei capitoli che seguono troverai:

- **Capitolo 1: Microcontrollori**  
Introduzione ai microcontrollori, numero di bit della CPU, indirizzamento e mappa di memoria, alternate functions e loro importanza, sensori e attuatori, segnali digitali e analogici, conversione A/D e protocolli di comunicazione (SPI, I2C, Serial).
- **Capitolo 2: La scheda Nucleo F401RE**  
Naming convention STM32, funzionalità hardware della Nucleo-F401RE, etichette dei pin, installazione e configurazione di STM32CubeIDE e introduzione a CubeMX.
- **Capitolo 3: GPIO output**  
Teoria sul funzionamento di un GPIO, configurazione del progetto, compilazione e programmazione, gestione del LED con lampeggio.
- **Capitolo 4: La comunicazione seriale**  
Teoria sul funzionamento della comunicazione seriale, configurazione del progetto, compilazione e programmazione, gestione del LED attraverso un comando seriale.
- **Capitolo 5: GPIO input – polling**  
Lettura di un GPIO configurato in input, gestione del pulsante utente in polling, approfondimento sui resistori di pull-up/pull-down.
- **Capitolo 6: GPIO input – interrupt**  
Generazione e gestione di interrupt da pin digitali, configurazione e priorità degli interrupt.
- **Capitolo 7: Analog input – ADC**  
Teoria sulla conversione analogico/digitale, configurazione dell'ADC su STM32F401RE e principali applicazioni.
- **Capitolo 8: Timers**  
Concetti di base sui timer, configurazione e utilizzo pratico per generare eventi

temporizzati.

- **Capitolo 9: PWM**

Fondamenti della modulazione di larghezza di impulso, configurazione del PWM su STM32 e esempi applicativi.

# Capitolo 1 Microcontrollori

## Introduzione ai Microcontrollori

Un **microcontrollore** è un dispositivo elettronico integrato che combina un processore (CPU), memoria e periferiche di input/output (I/O) su un unico chip. I microcontrollori sono progettati per applicazioni embedded, ovvero sistemi elettronici pensati per svolgere compiti specifici.

Un microcontrollore è composto da:

- **CPU (Central Processing Unit):** esegue le istruzioni del programma. Tipicamente è basato su architetture come ARM Cortex, AVR, o RISC.
- **Memoria:**
  - **Flash:** Per immagazzinare il programma.
  - **RAM:** Per dati temporanei.
  - **EEPROM (opzionale):** Per dati permanenti e modificabili
- **Periferiche:**
  - **GPIO:** Pin di input/output generici per comunicare con dispositivi esterni.
  - **ADC/DAC:** Convertitori per segnali, da analogici a digitali e viceversa.
  - **Timer:** Per funzioni di temporizzazione e conteggio.
  - **Bus di comunicazione - UART, I2C, SPI, CAN:** Per comunicazioni interne o con dispositivi esterni.
- **Oscillatore:**
  - Genera il clock per sincronizzare le operazioni del microcontrollore.

Le caratteristiche sulle quali si confrontano i microcontrollori sono tipicamente:

- **Risoluzione:** Ossia il numero di bit della CPU (es. 8-bit, 16-bit, 32-bit).
- **Frequenza di clock:** Velocità operativa del microcontrollore, misurata in MHz o GHz.
- **Consumi energetici:** Fondamentali per applicazioni a batteria o IoT.

## Il numero di bit della CPU

Il **numero di bit della CPU** di un microcontrollore (ad esempio, 8-bit, 16-bit, 32-bit) indica la dimensione dei dati che questo può elaborare in un singolo ciclo di istruzione, così come la larghezza dei registri interni e, in genere, l'architettura dell'unità di calcolo. È un parametro fondamentale che influenza sulle prestazioni, sulle capacità e sull'efficienza del microcontrollore.

Questo numero rappresenta:

- **Dimensione delle operazioni:**
  - Una CPU a **8-bit** può elaborare numeri o istruzioni di massimo 8 bit (1 byte) alla volta.
  - Una CPU a **16-bit** o **32-bit** può elaborare numeri o istruzioni più grandi (fino a 16 o 32 bit), risultando più efficiente per calcoli complessi.
- **Aampiezza dei registri:**
  - I registri interni della CPU hanno la stessa larghezza del numero di bit della CPU.
  - Ad esempio, una CPU a 32-bit avrà registri di 32 bit per memorizzare dati e indirizzi.
- **Gestione degli indirizzi di memoria:** Una CPU con un numero maggiore di bit può gestire un intervallo di indirizzi più ampio. Ad esempio, una CPU a 8 bit può indirizzare

$$2^8 = 256$$

locazioni di memoria, mentre una CPU a 32 bit può indirizzare

$$2^{32} = 4.294.967.296$$

cioè circa 4 miliardi di locazioni.

La seguente tabella riporta un confronto tra le varie architetture

Caratteristica	8-bit	16-bit	32-bit
Dimensione dei dati	Fino a 8 bit	Fino a 16 bit	Fino a 32 bit
Indirizzi di memoria	256 locazioni	65,536 locazioni	4 miliardi di locazioni
Efficienza nei calcoli	Limitata	Migliore	Ottima
Esempi di uso	Sistemi semplici	Automazione base	Applicazioni avanzate

Ad esempio, un microcontrollore a 8 bit, come l'**ATmega328** (usato in Arduino Uno), è ideale per compiti semplici come accendere un LED o leggere un sensore. Tuttavia, un microcontrollore a 32 bit, come lo **STM32F401RE**, è più adatto per applicazioni complesse, come il controllo di motori o l'elaborazione di segnali in tempo reale.

In sintesi, il numero di bit della CPU è un parametro cruciale che determina la capacità di calcolo, la velocità e le applicazioni per cui il microcontrollore è più adatto.

## Cosa indirizza un microcontrollore?

Quando si parla di **indirizzamento** in un microcontrollore, si fa riferimento alla capacità della CPU di accedere a specifiche locazioni di memoria. Il numero di bit utilizzati per l'indirizzamento determina la quantità di memoria che può essere indirizzata, ovvero "raggiunta" dalla CPU. La memoria di un microcontrollore può essere:

- **Memoria programma (Flash):** Contiene il codice del programma eseguibile. La CPU deve poter accedere alle istruzioni per eseguire il codice, quindi utilizza il sistema di indirizzamento per individuare la posizione di ciascuna istruzione.
- **Memoria dati (RAM):** Utilizzata per memorizzare dati temporanei, variabili, e stack. L'indirizzamento permette alla CPU di leggere e scrivere dati in specifiche locazioni di RAM.
- **Memoria periferica (I/O):** Include registri delle periferiche come UART, ADC, GPIO, ecc. Questi registri sono mappati in una porzione dell'indirizzo di memoria, e la CPU li gestisce come parte del sistema di indirizzamento.
- **EEPROM (opzionale):** Per la memorizzazione di dati permanenti o configurazioni. Anche questa memoria può essere indirizzata dalla CPU.

Un microcontrollore come l'**STM32F401RE** ha una CPU a 32 bit, il che significa che il suo bus di indirizzi è 'largo' 32 bit. Questo gli permette di generare indirizzi compresi tra **0x00000000** e **0xFFFFFFFF**, cioè  $2^{32}$  locazioni di memoria, per un totale di **4 GB** indirizzabili. Tuttavia, la memoria fisica disponibile è molto inferiore a questo limite teorico.

- **Flash Memory (Program Memory):** 512 KB, da 0x08000000 a 0x0807FFFF.
- **RAM (Data Memory):** 96 KB, da 0x20000000 a 0x20017FFF.
- **Periferiche:** Mappate in un range specifico (ad esempio, GPIO, UART).
- **Memoria di sistema:** Un'area riservata per il bootloader e altre funzioni di sistema.

## Come funziona l'indirizzamento

La CPU accede a queste memorie utilizzando indirizzi. Ogni indirizzo identifica una specifica locazione o registro. Vediamo alcuni esempi passo per passo.

Supponiamo che il microcontrollore debba leggere una variabile memorizzata nella RAM. La variabile si trova a un indirizzo specifico, ad esempio 0x20000004.

- **Generazione dell'indirizzo:** La CPU calcola o riceve l'indirizzo 0x20000004 e lo invia al bus di indirizzi.
- **Accesso alla memoria:** L'unità di controllo verifica che l'indirizzo rientri nel range della RAM (da 0x20000000 a 0x20017FFF). Se l'indirizzo è valido, accede alla locazione di memoria corrispondente.
- **Recupero del dato:** La CPU legge il dato presente in 0x20000004 e lo memorizza in un registro interno per ulteriori elaborazioni.

Ora supponiamo che il microcontrollore debba scrivere un'istruzione nella Flash Memory, all'indirizzo 0x08000010:

- **Indirizzo generato:** La CPU invia l'indirizzo 0x08000010 al bus di indirizzi.
- **Accesso alla Flash:** Il sistema controlla che l'indirizzo sia nel range della Flash (da 0x08000000 a 0x0807FFFF). Se sì, consente la scrittura (sempre che la Flash sia in modalità scrivibile).
- **Scrittura del dato:** L'istruzione viene scritta nella locazione indicata.

Supponiamo invece che voglia controllare una periferica, per esempio un LED controllato da un pin GPIO, e il registro del GPIO si trovi all'indirizzo 0x40020014.

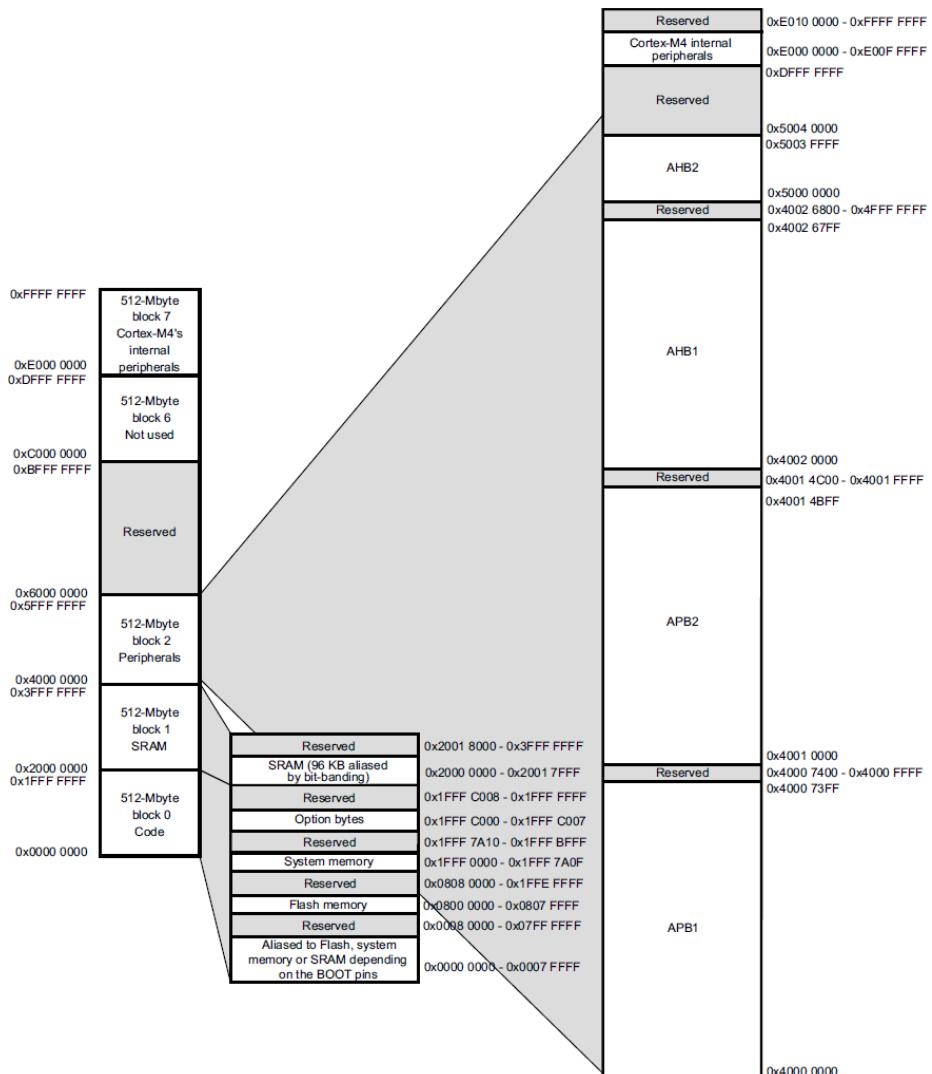
- **Indirizzo del registro GPIO:** La CPU genera l'indirizzo 0x40020014, che corrisponde a un registro del GPIO per impostare lo stato di un pin.
- **Scrittura nel registro:** La CPU scrive un valore (ad esempio, 1 per accendere il LED) nel registro.

## La mappa di memoria

Nel STM32F401RE, le diverse aree di memoria sono mappate su un range specifico:

Range Indirizzi	Tipo di Memoria	Descrizione
0x08000000 - 0x0807FFFF	Flash Memory	Program Memory (512 KB)
0x20000000 - 0x20017FFF	RAM	Data Memory (96 KB)
0x40000000 - 0x500607FF	Periferiche	Registri delle periferiche (GPIO, UART, ecc.)
0x1FFF0000 - 0x1FFF77FF	System Memory	Bootloader e altre funzioni di sistema

La seguente figura, estratta dal datasheet del microcontrollore STM32F401RE, esplora in dettaglio la mappa di memoria del dispositivo.



In pratica, l'indirizzamento permette alla CPU di gestire tutti i componenti (RAM, Flash, periferiche) utilizzando un unico sistema di indirizzi. Ogni operazione è guidata dalla logica del microcontrollore e mappata su una porzione del bus di indirizzi, rendendo l'accesso alla memoria e ai dispositivi semplice e uniforme.

## Applicazioni

Le applicazioni dei Microcontrollori più diffuse sono:

- **Domotica:** Controllo di luci, termostati, serrature intelligenti.
- **Automotive:** Sistemi di controllo motore, airbag, ABS.
- **Industria:** Automazione di processi e macchinari.
- **IoT (Internet of Things):** Dispositivi connessi per raccolta e analisi dati.
- **Elettronica di consumo:** Elettrodomestici, giocattoli interattivi, orologi intelligenti.

I microcontrollori sono la base dell'elettronica embedded moderna, con applicazioni che spaziano dalla domotica all'industria. Comprendere la loro struttura e programmazione è fondamentale per sviluppare soluzioni tecnologiche innovative.

## Alternate Functions nei Microcontrollori

I microcontrollori moderni sono dispositivi estremamente versatili, dotati di numerosi pin che possono essere configurati per svolgere diverse funzioni. Queste configurazioni multiple, spesso chiamate *alternate functions*, consentono al microcontrollore di adattarsi a una vasta gamma di applicazioni, ottimizzando l'utilizzo delle risorse hardware disponibili.

### Cosa Sono le Alternate Functions

Un pin di un microcontrollore non è limitato a una sola funzione. Oltre alla funzione di base (tipicamente un ingresso o un'uscita digitale), molti pin possono essere utilizzati per altre funzioni, come:

- **Ingressi/Uscite analogici** (ADC o DAC)
- **Comunicazioni seriali** (UART, SPI, I2C, CAN, ecc.)
- **PWM (Pulse Width Modulation)** per controllare motori o LED
- **Ingressi di timer**
- **Funzioni speciali del microcontrollore**, come clock o debug

Queste funzioni alternative sono configurabili attraverso registri di controllo interni del microcontrollore.

### Perché sono importanti

Le alternate functions consentono di:

- **Ottimizzare l'uso dei pin:** Un microcontrollore con un numero limitato di pin può offrire molte funzionalità grazie alla configurazione dinamica.
- **Ridurre le dimensioni del sistema:** Implementando più funzionalità su meno pin, si riducono le dimensioni del PCB e i costi di produzione.
- **Fornire flessibilità progettuale:** Lo sviluppatore può scegliere come configurare i pin per adattarsi meglio all'applicazione.

### Come Funzionano

Le alternate functions sono gestite tramite:

- **Registri di configurazione GPIO:** Ogni pin ha associati uno o più registri di configurazione. Questi registri specificano:
  - Modalità del pin (digitale, analogico, ecc.)
  - Velocità del pin
  - Pull-up/pull-down resistors
  - Funzione alternativa selezionata
- **Mappatura delle Funzioni Alternative:** Ogni pin ha una tabella di mappatura che specifica quali funzioni alternate sono disponibili. Ad esempio, nel datasheet del microcontrollore viene fornita una matrice che elenca le funzioni alternate di ogni pin.

- **Multiplexer Interni:** All'interno del microcontrollore, i pin sono collegati a un multiplexer hardware. La configurazione dei registri seleziona quale funzione è abilitata per quel pin.

Le alternate functions rappresentano un aspetto fondamentale per sfruttare al meglio un microcontrollore. Una corretta configurazione è essenziale per garantire il funzionamento ottimale dell'applicazione. Lo sviluppatore deve sempre consultare il datasheet e utilizzare gli strumenti software (come HAL o driver specifici) per configurare queste funzionalità in modo efficiente.

## Alternate functions del microcontrollore STM32F401RE

Per comodità del lettore, viene riportata di seguito la tabella di tutte le alternate functions offerte da ciascun pin del microcontrollore STM32F401RE. Le tabelle sono estratte dal datasheet che si consiglia di consultare.

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				
PortA	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_ SMBAS	-	-	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_SD	-	I2C3_SDA	-	-	-	-	EVENT OUT
	PB5	-	-	TIM3_CH2	-	I2C1_SMB	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/I2S2_WS	-	-	-	-	-	-	SDIO_D5	-	EVENT OUT
	PB10	-	TIM2_CH3	-	-	I2C2_SCL	SPI2_SCK/I2S2_CK	-	-	-	-	-	-	-	-	EVENT OUT
	PB12	-	TIM1_BKIN	-	-	I2C2_SMB	SPI2_NSS/I2S2_WS	-	-	-	-	-	-	-	-	EVENT OUT
	PB13	-	TIM1_CH1N	-	-	-	SPI2_SCK/I2S2_CK	-	-	-	-	-	-	-	-	EVENT OUT
	PB14	-	TIM1_CH2N	-	-	-	SPI2_MISO	I2S2ext_SD	-	-	-	-	-	-	-	EVENT OUT
	PB15	RTC_REFN	TIM1_CH3N	-	-	-	SPI2_MOSI/I2S2_SD	-	-	-	-	-	-	-	-	EVENT OUT
Port C	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 E	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					
	PE0	-	-	TIM4_ETR	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT	
	PE1	-	TIM1_CH2N	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT	
	PE2	TRACECLK	-	-	-	-	SPI4_SCK	-	-	-	-	-	-	-	-	EVENT OUT	
	PE3	TRACEDD0	-	-	-	-	-	-	-	-	-	-	-	-	-	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

Il *Physical Computing* è l'intersezione tra il mondo fisico e il digitale, dove i dispositivi elettronici, come i microcontrollori, vengono utilizzati per interagire con l'ambiente attraverso sensori, attuatori e dispositivi di input/output. Questo approccio è fondamentale per sviluppare applicazioni interattive, prototipi innovativi e soluzioni IoT.

I microcontrollori sono componenti essenziali del physical computing grazie alla loro capacità di:

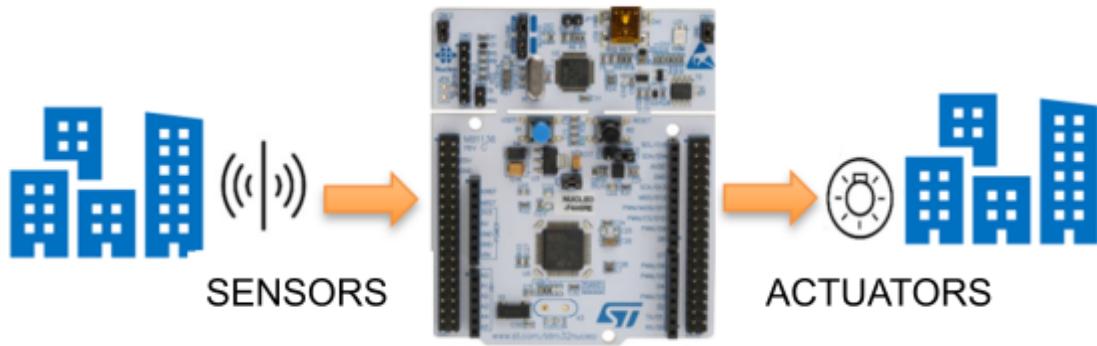
- **Rilevare l'ambiente** attraverso sensori (es. temperatura, luce, pressione).
- **Elaborare i dati** provenienti dai sensori.
- **Interagire con l'ambiente** tramite attuatori (es. motori, LED, speaker).
- **Comunicare** con altri dispositivi tramite protocolli standard (es. I2C, SPI, UART, Wi-Fi).

Un sistema di physical computing si basa su tre componenti principali:

- **Input:**
  - Sensori per rilevare dati fisici o ambientali (es. temperatura, umidità, luce).
  - Dispositivi di input come pulsanti, potenziometri, joystick o sensori tattili.
- **Elaborazione:**
  - Microcontrollore che elabora i dati e decide le azioni da intraprendere.
  - Algoritmi programmati per analizzare e rispondere agli input ricevuti.
- **Output:**
  - Attuatori per rispondere agli input (es. motori, elettrovalvole, relè).
  - Feedback visivo o acustico (es. LED, display LCD, buzzer).

Esempi di Progetti di Physical Computing sono:

- **Stazione Meteo IoT:**
  - **Input:** Sensori di temperatura, umidità e pressione.
  - **Elaborazione:** Calcolo dei dati ambientali e invio al cloud.
  - **Output:** Visualizzazione su display LCD o dashboard online.
- **Robot Mobile:**
  - **Input:** Sensori di distanza (es. ultrasuoni) e pulsanti di controllo.
  - **Elaborazione:** Algoritmi di evitamento ostacoli.
  - **Output:** Controllo dei motori per il movimento.
- **Smart Home:**
  - **Input:** Sensori PIR per rilevare movimento e interruttori smart.
  - **Elaborazione:** Logica di accensione/spegnimento automatica.
  - **Output:** Controllo di luci, tapparelle o dispositivi connessi.



Dunque i sensori percepiscono l'ambiente e forniscono queste informazioni al microcontrollore, che in base al programma in esecuzione, può eseguire azioni tramite gli attuatori.

Il physical computing con i microcontrollori offre infinite possibilità per creare soluzioni innovative che interagiscono con il mondo fisico. Comprendere i principi base e sperimentare con progetti pratici è il modo migliore per sviluppare competenze in questo campo.

## Sensori

Un sensore è un dispositivo che rileva cambiamenti fisici o chimici nell'ambiente e li converte in segnali elettrici che possono essere letti da un sistema elettronico, come un microcontrollore. I sensori vengono utilizzati in una vasta gamma di applicazioni, dall'automazione industriale all'Internet delle Cose (IoT), per raccogliere informazioni sull'ambiente circostante.

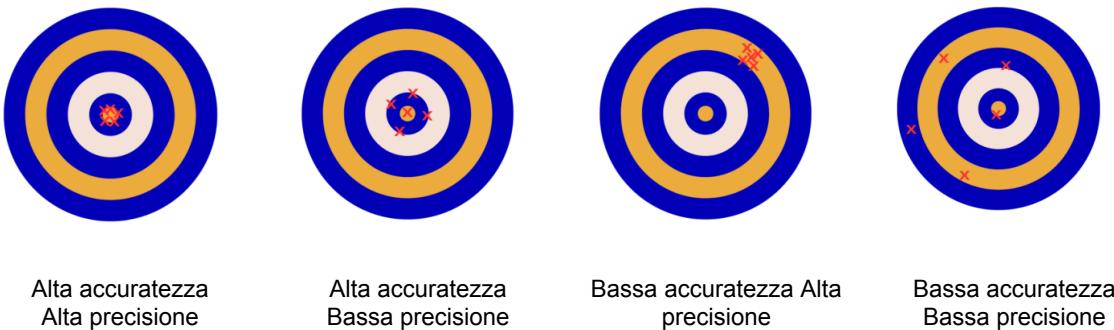


Un sensore ha le seguenti caratteristiche principali:

- **Grandezza Misurata:** È la variabile fisica o chimica che il sensore rileva, come temperatura, umidità, pressione, luce, posizione, gas o forza.
- **Uscita:** La grandezza rilevata viene convertita in un segnale elettrico, che può essere:
  - **Analogico:** Un segnale continuo (ad esempio, un valore di tensione proporzionale alla grandezza misurata).
  - **Digitale:** Un segnale discreto, spesso trasmesso utilizzando protocolli come I2C, SPI o UART.

- **Intervallo (Range):** L'intervallo di valori che il sensore può misurare.
- **Accuratezza e Risoluzione:**
  - **Accuratezza:** La capacità del sensore di fornire un valore vicino a quello reale.
  - **Precisione:** la capacità di indicare ripetutamente uno stesso valore di misura, indipendentemente dal fatto che corrisponda o meno al valore vero.
  - **Risoluzione:** Il livello di dettaglio con cui il sensore rileva le variazioni.
- **Sensibilità:** La capacità del sensore di rispondere a piccole variazioni nella grandezza misurata.

L'esempio del bersaglio consente di comprendere facilmente i significati di accuratezza e precisione di un sensore



## Esempio: Sensore di Temperatura sulla Scheda di Espansione IKS4A1

La **X-NUCLEO-IKS4A1** è una scheda di espansione progettata per le piattaforme STM32 Nucleo, dotata di sensori avanzati per il rilevamento del movimento e il monitoraggio ambientale. Tra i sensori integrati si trova lo STTS22H, un sensore digitale ad alta precisione per la misurazione della temperatura.

Le caratteristiche Tecniche del STTS22H:

- **Intervallo di Misurazione della Temperatura:** da -40 °C a +125 °C.
- **Accuratezza:**  $\pm 0,5$  °C entro l'intervallo operativo standard.
- **Risoluzione:** 16 bit (fino a 0,0625 °C per incremento).
- **Interfaccia di Comunicazione:**
  - I<sup>2</sup>C con indirizzo slave configurabile.
  - Supporta velocità fino a 1 MHz (modalità Fast Mode Plus di I<sup>2</sup>C).
- **Consumo Energetico:**
  - **1,75 µA** in modalità attiva (a 1 Hz).
  - **0,5 µA** in modalità standby.

Il sensore STTS22H misura la temperatura utilizzando un sistema a giunzione diodi. Le variazioni di temperatura influenzano il comportamento elettrico del diodo, e il sensore elabora queste variazioni per fornire valori digitali altamente precisi.

Il sensore STTS22H si integra facilmente con qualsiasi scheda STM32 Nucleo grazie alla compatibilità con il layout Arduino Uno R3 presente sulla X-NUCLEO-IKS4A1. La comunicazione avviene tramite bus I<sup>2</sup>C.

Dunque in sintesi, il sensore percepisce la temperatura attraverso una variazione del comportamento elettrico del diodo e mette a disposizione il dato della temperatura attraverso un protocollo I<sup>2</sup>C.

## Gli attuatori

Gli **attuatori** sono dispositivi che trasformano un segnale di controllo, generalmente di natura elettrica, in un'azione fisica. Sono utilizzati per compiere movimenti, applicare forze o generare effetti specifici come luce, calore o suono. Gli attuatori rappresentano la parte "attiva" di un sistema di automazione o controllo, interagendo direttamente con l'ambiente circostante.

Gli attuatori possono essere classificati in base a:

- **Energia utilizzata:**
  - **Elettrici:** Motori, elettromagneti, solenoidi.
  - **Idraulici:** Pistoni e cilindri alimentati da fluidi in pressione.
  - **Pneumatici:** Attuatori alimentati da aria compressa.
- **Tipo di movimento:**
  - **Lineari:** Producono un movimento rettilineo (es. pistoni).
  - **Rotativi:** Producono un movimento rotatorio (es. motori DC, stepper).
- **Funzione svolta:**
  - **Di posizionamento:** Per regolare la posizione di un oggetto.
  - **Di forza:** Per applicare una forza specifica.

## Attuatori Elettrici

Gli attuatori elettrici utilizzano l'energia elettrica per generare movimento o effetti fisici. Esempio di questo tipo di attuatori sono:

- **Motori DC (corrente continua):**



- Movimento rotatorio continuo.
- Semplici da controllare tramite segnali PWM.
- Utilizzati in robotica, veicoli, elettrodomestici.

- **Motori Stepper:**



- Movimento rotatorio a passi discreti.
- Ideali per applicazioni che richiedono un controllo preciso della posizione, come stampanti 3D o macchine CNC.

- **Servomotori:**

- Forniscono controllo preciso di posizione, velocità e coppia.
- Combinano un motore DC, un riduttore e un sistema di controllo.
- Utilizzati in bracci robotici e automazione.



- **Solenoidi:**



- Generano un movimento lineare quando alimentati.
- Utilizzati in serrature elettroniche, valvole e meccanismi di blocco.

## Attuatori Idraulici

- **Caratteristiche:**
  - Elevata forza e robustezza.
  - Ideali per applicazioni industriali pesanti (es. macchinari da costruzione).
- **Esempi:**
  - Cilindri idraulici per sollevamento e compressione.
  - Sistemi di sterzo idraulico nei veicoli.

## Attuatori Pneumatici



Gli attuatori pneumatici utilizzano aria compressa per generare movimento.

- **Caratteristiche:**
  - Veloci, economici e sicuri.
  - Adatti per ambienti industriali e sistemi automatizzati.
- **Esempi:**
  - Pistoni pneumatici per macchinari di assemblaggio.
  - Valvole controllate pneumaticamente.

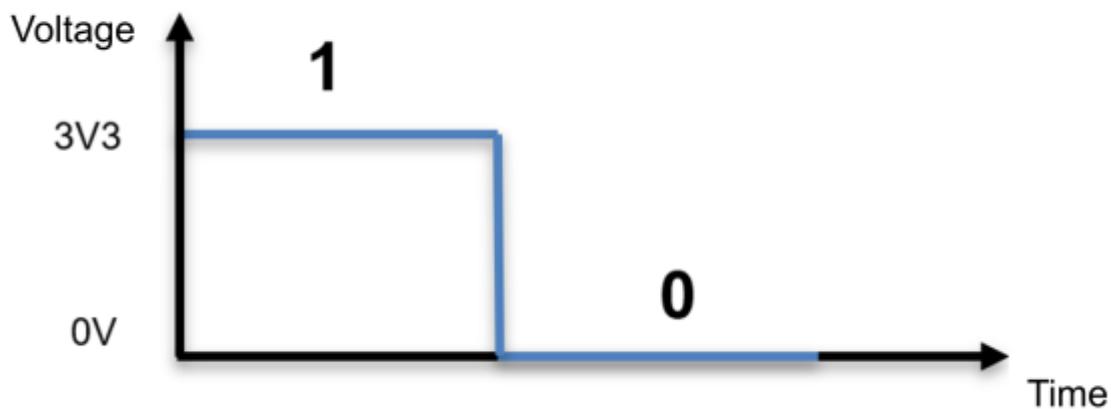
## Attuatori Termici

Utilizzano variazioni di temperatura per generare movimento o cambiamenti di stato.

- **Esempi:**
  - Attuatori a cera per termostati.
  - Elementi piezoelettrici che deformano in risposta al calore.

## I segnali digitali

Un segnale digitale è un tipo di segnale che rappresenta le informazioni utilizzando valori discreti, tipicamente binari (0 e 1). A differenza dei segnali analogici, che variano in modo continuo, un segnale digitale assume solo un numero finito di stati, rendendolo più facile da elaborare e trasmettere nei sistemi elettronici.



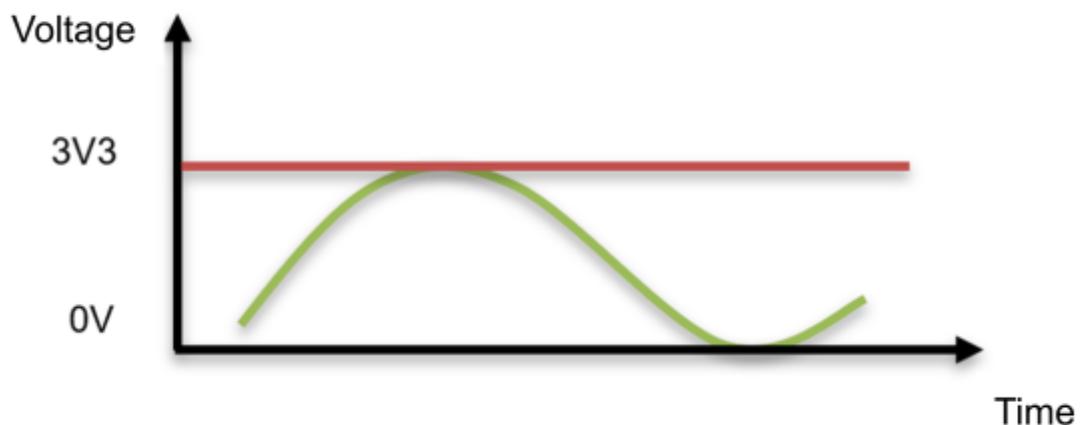
**Le caratteristiche principali sono:**

- **Valori discreti:** Di solito rappresentati da livelli di tensione, ad esempio:
  - 0: Tensione bassa (es. 0V).
  - 1: Tensione alta (es. 3,3V o 5V).
- **Immunità al rumore:** I segnali digitali sono meno sensibili al rumore rispetto ai segnali analogici, poiché il sistema rileva solo due stati distinti.
- **Facilità di elaborazione:** Sono utilizzati nei sistemi digitali, come microcontrollori e processori, dove è possibile eseguire operazioni logiche e aritmetiche con precisione.

I segnali digitali sono fondamentali per la comunicazione e l'elaborazione nei sistemi elettronici moderni, inclusi computer, dispositivi IoT e reti di telecomunicazione.

## I segnali analogici

Un segnale analogico è un tipo di segnale che varia continuamente nel tempo, rappresentando informazioni attraverso un intervallo infinito di valori possibili all'interno di un determinato intervallo. È comunemente utilizzato per rappresentare fenomeni fisici come il suono, la luce o la temperatura.



Le caratteristiche principali sono:

- **Continuità:** Il valore del segnale può assumere qualsiasi valore all'interno dell'intervallo specificato.
- **Sensibilità al rumore:** I segnali analogici sono più soggetti a rumore e distorsione rispetto ai segnali digitali.
- **Rappresentazione naturale:** Sono ideali per rappresentare fenomeni fisici che variano in modo continuo.

Esempi di segnali analogici includono la voce catturata da un microfono, le onde luminose rilevate da un sensore ottico e la tensione in un circuito variabile.

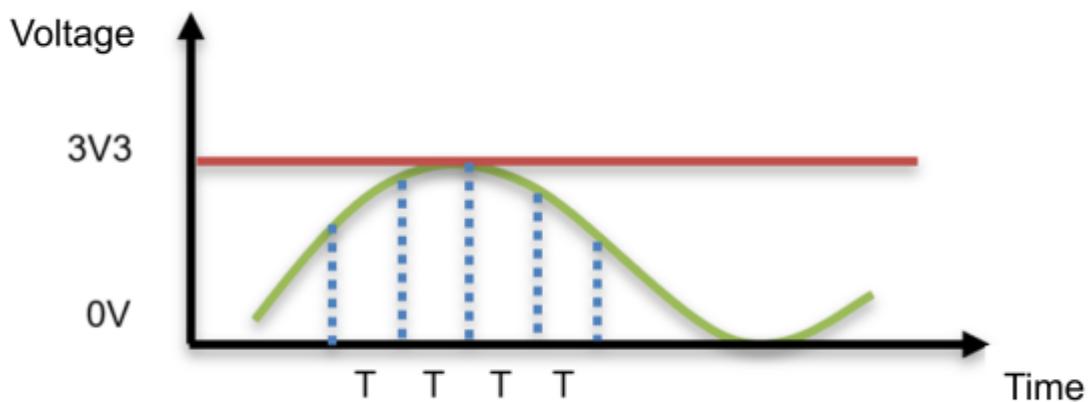
## Conversione Analogico/Digitale

A questo punto risulta chiaro che il microcontrollore lavora nel ‘mondo digitale’, ma interagisce con un ‘mondo analogico’. Per poter consentire questa interazione è necessario disporre di dispositivi utili alla conversione dei segnali.

La conversione Analogico-Digitale (ADC) è il processo di trasformazione di un segnale analogico continuo in un segnale digitale discreto, ossia che assume un numero finito di valori che vengono codificati, che può essere elaborato da sistemi elettronici digitali come microcontrollori o computer. Questo processo è essenziale per acquisire e analizzare informazioni provenienti dal mondo fisico, come temperatura, luce o suono.

Il processo di conversione ADC comprende tre fasi principali:

- **Campionamento (Sampling):**
  - Il segnale analogico viene misurato a intervalli di tempo regolari, definiti dalla frequenza di campionamento.
  - La frequenza di campionamento deve essere almeno il doppio della frequenza massima del segnale, secondo il teorema di Nyquist, per evitare aliasing.
- **Quantizzazione (Quantization):** I valori continui del segnale campionato vengono mappati su un numero finito di livelli discreti, determinati dalla risoluzione dell'ADC. Ad esempio, un ADC a 12 bit suddivide l'intervallo analogico in:  
$$2^{12} = 4096 \text{ livelli}$$
- **Codifica (Encoding):** Ogni valore quantizzato viene rappresentato come un numero binario, leggibile dal sistema digitale.



Le caratteristiche principali di un ADC sono:

- **Risoluzione:** Indica il numero di bit utilizzati per rappresentare i valori digitali. Ad esempio, un ADC a 12 bit ha una risoluzione pari a 1/4096 dell'intervallo totale.

$$Step = \frac{V_{ref}}{2^{12}} = \frac{V_{ref}}{4096}$$

- **Intervallo di tensione (Voltage Range):** L'intervallo di tensione che l'ADC può misurare (es. 0-3,3V o ±5V).
- **Accuratezza:** La differenza tra il valore reale del segnale analogico e il valore digitale convertito.
- **Frequenza di campionamento (Sampling Rate):** La velocità con cui l'ADC può misurare il segnale, espressa in campioni al secondo (SPS).
- **Errore di quantizzazione (Quantization Error):** La differenza introdotta durante la conversione tra il segnale analogico continuo e il segnale digitale discreto.

## Esempio pratico: ADC su STM32F401RE

Il microcontrollore STM32F401RE è dotato di un ADC a 12 bit, che consente di convertire un segnale analogico (ad esempio una tensione tra 0 e 3,3V) in un valore digitale compreso tra 0 e 4095.

La frequenza di campionamento è configurabile e dipende dalla frequenza del clock dell'ADC.

Se la tensione analogica in ingresso è di 1,65V e la tensione di riferimento massima è 3,3V, il valore digitale corrispondente si calcola come segue:

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

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

Il valore digitale **2048** rappresenta **1,65V** nel sistema.

## Le principali applicazioni dell'ADC

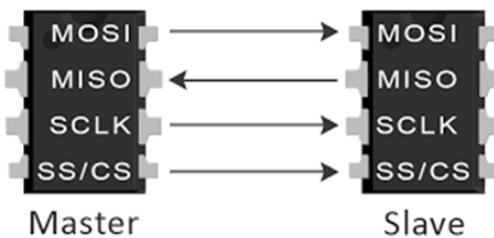
- **Audio:** Convertire segnali sonori analogici in dati digitali per registrazione o elaborazione.
- **Sensori:** Misurare grandezze fisiche come temperatura, luce, pressione o altre variabili.
- **Imaging:** Convertire segnali provenienti da sensori ottici in immagini digitali.
- **Automazione:** Controllare sistemi basati su feedback analogico, come motori o sistemi HVAC (Heating, Ventilation & Air Conditioning).

# I protocolli di comunicazione

La comunicazione con altri dispositivi connessi al microcontrollore, avviene mediante protocolli dedicati che definiscono le modalità di connessione e le regole di comunicazione. Di seguito esaminiamo i principali protocolli,

## SPI - Serial Peripheral Interface

Il protocollo SPI (Serial Peripheral Interface) è un metodo di comunicazione seriale sincrona ad alta velocità utilizzato per lo scambio di dati tra microcontrollori e dispositivi periferici come sensori, display o memorie. Funziona con una configurazione master-slave, con un master e uno o più slave.



Le caratteristiche principali sono:

- **Comunicazione sincrona:** Un clock (SCLK) sincronizza il trasferimento dei dati tra master e slave.
- **Full-duplex:** I dati vengono trasmessi e ricevuti contemporaneamente su linee separate.
- **Alta velocità:** Supporta una comunicazione più rapida rispetto ad altri protocolli seriali come I<sup>2</sup>C o UART.
- **Topologia semplice:** Relativamente facile da implementare, ma richiede più cavi rispetto ad altri protocolli.

Le principali linee SPI sono:

- **SCLK (Serial Clock):** Generato dal master, sincronizza il trasferimento dei dati.
- **MOSI (Master Out Slave In):** Linea su cui il master invia dati allo slave.
- **MISO (Master In Slave Out):** Linea su cui lo slave invia dati al master.
- **SS/CS (Slave Select/Chip Select):** Linea dedicata per selezionare quale slave è attivo.

Funzionamento in base ai seguenti passaggi:

- **Configurazione iniziale:** Il master imposta i parametri principali, come velocità del clock, polarità del clock (CPOL), fase del clock (CPHA) e ruoli delle linee. Ogni slave è collegato al master tramite linee dedicate (MOSI, MISO, SCLK), ma ogni slave ha una propria linea SS/CS.

- **Selezione dello slave:** Il master porta la linea SS/CS dello slave target a livello basso (nel caso di logica attiva bassa). Gli altri slave ignorano la comunicazione mantenendo alta la propria linea SS/CS.
- **Scambio dati:** Il master genera segnali di clock (SCLK) e trasferisce dati tramite MOSI. Lo slave risponde simultaneamente inviando dati tramite MISO. Il trasferimento dei dati avviene un bit alla volta, sincronizzato con il segnale SCK.
- **Fine comunicazione:** Il master porta la linea SS/CS a livello alto, segnalando la fine della comunicazione con quello slave.

Vantaggi del protocollo SPI:

- **Alta velocità:** Velocità di trasferimento che possono raggiungere decine di MHz.
- **Full-duplex:** Permette il trasferimento bidirezionale simultaneo dei dati.
- **Hardware semplice:** Facile da implementare con linee dedicate.

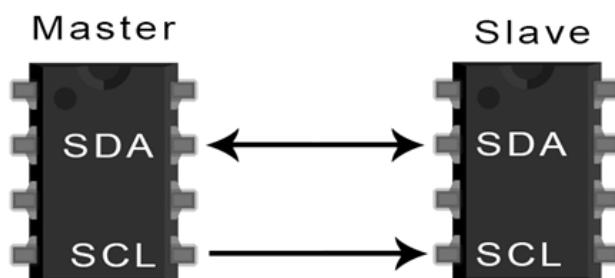
Gli svantaggi del protocollo sono invece:

- **Elevato utilizzo di pin:** Ogni slave richiede una linea SS/CS dedicata, che può diventare problematica con molti dispositivi.
- **Assenza di indirizzamento o controllo errori integrati:** Richiede gestione aggiuntiva tramite software.

SPI è ideale per applicazioni che richiedono una comunicazione rapida e un numero limitato di periferiche, come sensori o dispositivi di memoria ad alta velocità.

## I<sup>2</sup>C

Il protocollo I<sup>2</sup>C (Inter-Integrated Circuit) è un protocollo di comunicazione seriale sincrona, multi-master e multi-slave, progettato per la comunicazione a bassa velocità tra circuiti integrati. È ampiamente utilizzato per collegare microcontrollori a sensori, display e altri dispositivi periferici.



Le caratteristiche principali sono:

- **Comunicazione sincrona:** Il trasferimento dei dati è sincronizzato tramite una linea di clock (SCL).
- **Comunicazione a due fili:** Richiede solo due linee, riducendo l'uso di pin rispetto ad altri protocolli.
- **Comunicazione basata su indirizzi:** Ogni slave ha un indirizzo unico, permettendo la comunicazione con molti dispositivi utilizzando le stesse linee.

Le principali linee I<sup>2</sup>C sono:

- **SCL (Serial Clock Line):** Il segnale di clock generato dal master per sincronizzare il trasferimento dei dati.
- **SDA (Serial Data Line):** Linea bidirezionale utilizzata per inviare e ricevere dati.
  - Entrambe le linee sono open-drain e richiedono resistori di pull-up per mantenere uno stato alto di default.

Il funzionamento è definito dai seguenti passi:

- **Configurazione iniziale:** Il master configura la velocità del clock e definisce i parametri di comunicazione, come la modalità di indirizzamento (7-bit o 10-bit).
- **Condizione di start:** Il master genera una condizione di start abbassando la linea SDA mentre SCL rimane alta, segnalando l'inizio di un frame di comunicazione.
- **Indirizzamento:**
  - Il master invia l'indirizzo dello slave target, seguito da un bit di lettura/scrittura che indica il tipo di operazione.
  - Tutti i dispositivi sul bus ascoltano, ma solo lo slave indirizzato risponde abbassando SDA (acknowledge o ACK).
- **Trasferimento dati:**
  - Master e slave scambiano dati in pacchetti da 8 bit (byte), ciascuno seguito da un acknowledgment (ACK) o da un no-acknowledgment (NACK).
  - Gli impulsi di clock (SCL) controllano quando i dati vengono letti o scritti.
- **Condizione di stop:** Il master genera una condizione di stop rilasciando SDA a livello alto mentre SCL rimane alto, segnalando la fine della comunicazione.

Il protocollo I<sup>2</sup>C offre i seguenti vantaggi:

- **Basso utilizzo di pin:** Solo due linee sono necessarie per la comunicazione, indipendentemente dal numero di dispositivi.
- **Capacità multi-master:** Più master possono coesistere sullo stesso bus.
- **Indirizzamento:** I dispositivi possono condividere lo stesso bus grazie a indirizzi unici, senza linee aggiuntive.

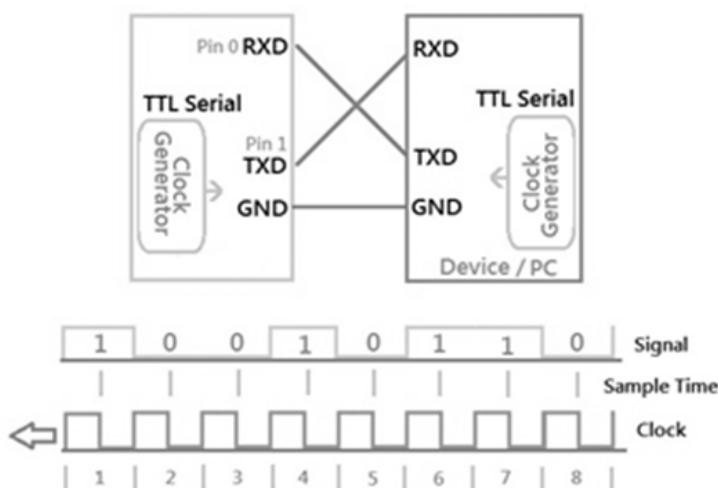
Gli svantaggi sono invece:

- **Più lento rispetto a SPI:** Velocità limitate a pochi MHz, a seconda dell'implementazione.
- **Lunghezza limitata del bus:** A causa di problemi di capacità, il bus è adatto solo per distanze brevi.
- **Complessità:** Richiede la gestione di acknowledgment e indirizzamento, aumentando il carico software.

I<sup>2</sup>C è ideale per applicazioni che richiedono velocità di trasferimento da basse a moderate e la connessione di più dispositivi utilizzando un numero minimo di fili, come nei sistemi embedded e nelle applicazioni IoT. La sua semplicità e scalabilità lo rendono una scelta popolare per la comunicazione tra componenti su una scheda PCB.

## Serial

Il protocollo seriale (UART - Universal Asynchronous Receiver-Transmitter) è un protocollo di comunicazione semplice e ampiamente utilizzato che consente il trasferimento di dati tra dispositivi, come microcontrollori, sensori o computer, utilizzando una comunicazione seriale asincrona. Funziona senza un segnale di clock e si basa su una configurazione predefinita per la sincronizzazione.



Le caratteristiche principali sono:

- **Comunicazione asincrona:** Non richiede un segnale di clock da un master ad uno slave; la sincronizzazione avviene tramite bit di start e stop. Entrambe i dispositivi devono conoscere la velocità di comunicazione.
- **Full-duplex:** I dati possono essere inviati e ricevuti simultaneamente su linee separate.

- **Comunicazione punto a punto:** Tipicamente utilizzato per collegare direttamente due dispositivi.
- **Impostazioni configurabili:** I parametri di comunicazione, come baud rate, parità, bit di dati e bit di stop, devono essere concordati da entrambi i dispositivi.

Le principali linee seriali sono:

- **TX (Transmit):** Linea su cui il dispositivo invia i dati.
- **RX (Receive):** Linea su cui il dispositivo riceve i dati.
- **Linee di controllo opzionali:** Come CTS (Clear to Send) e RTS (Request to Send) per il controllo di flusso hardware.

Il protocollo seriale funziona nelle seguenti modalità:

- **Configurazione:** Entrambi i dispositivi devono concordare sui parametri di comunicazione:
  - **Baud rate:** Velocità di comunicazione in bit al secondo (es. 9600 bps).
  - **Bit di dati:** Numero di bit in ogni frame di dati (es. 7 o 8 bit).
  - **Bit di parità:** Bit opzionale per il controllo degli errori (pari, dispari o nessuna parità).
  - **Bit di stop:** Numero di bit che segnano la fine di un frame di dati (es. 1 o 2 bit).
- **Framing dei dati:** Ogni byte di dati viene incapsulato in un frame così realizzato:
  - **Bit di start:** Un singolo bit (livello basso) che segnala l'inizio del frame.
  - **Bit di dati:** I dati reali (es. 8 bit).
  - **Bit di parità opzionale:** Per il controllo degli errori.
  - **Bit di stop:** Uno o più bit (livello alto) che segnano la fine del frame.
- **Trasmissione dei dati:**
  - Il trasmettitore invia i dati incapsulati frame per frame, bit per bit.
  - Il ricevitore campiona la linea dati alla velocità concordata (baud rate) per ricostruire i dati originali.
- **Gestione degli errori:** Possono verificarsi errori come quelli di framing o di parità, che devono essere gestiti dal dispositivo ricevente.

Il protocollo seriale offre i seguenti messaggi:

- **Semplicità:** Facile da implementare e richiede un hardware minimo.
- **Basso utilizzo di pin:** Solo due linee (TX e RX) sono necessarie per la comunicazione.
- **Asincrono:** Non richiede una linea di clock, semplificando il cablaggio.

Gli svantaggi sono invece:

- **Limitato a due dispositivi:** Connessione diretta tra soli due dispositivi (a meno che non si utilizzino multiplexer).
- **Nessun indirizzamento integrato:** Non può comunicare nativamente con più dispositivi.
- **Più lento dei protocolli sincroni:** A causa dell'overhead dei bit di start, stop e parità opzionali.

Il protocollo seriale è ideale per una comunicazione semplice e a bassa velocità tra due dispositivi. La sua semplicità e diffusione lo rendono una scelta comune per il debug, gli aggiornamenti firmware e la registrazione dei dati nei sistemi embedded.

# Ricapitoliamo

Abbiamo esplorato il mondo del **Physical Computing**, dove i microcontrollori rappresentano il cuore pulsante di sistemi in grado di connettere il mondo fisico e quello digitale. Attraverso sensori, attuatori e protocolli di comunicazione, i microcontrollori raccolgono informazioni dall'ambiente, le elaborano e rispondono con azioni precise.

## Punti chiave da ricordare:

- **Connessione con l'ambiente:**
  - **Sensori:** Dispositivi che percepiscono variabili fisiche (es. temperatura, luce) e le convertono in segnali elettrici. Questi segnali possono essere:
    - **Analogici:** Valori continui che richiedono conversione ADC per essere compresi dal microcontrollore.
    - **Digitali:** Valori discreti, spesso trasmessi tramite protocolli standard come I<sup>2</sup>C, SPI o UART.
  - **Attuatori:** Dispositivi che trasformano segnali elettrici in azioni fisiche, come movimento, luce o suono, chiudendo il ciclo di interazione con l'ambiente.
- **Protocolli di comunicazione:**
  - **SPI:** Ideale per connessioni ad alta velocità e full-duplex con periferiche come display e memorie.
  - **I<sup>2</sup>C:** Perfetto per sistemi multi-slave con un numero ridotto di fili.
  - **UART:** Un'opzione semplice e diffusa per la comunicazione punto a punto. Ogni protocollo ha caratteristiche uniche che lo rendono adatto a scenari specifici, assicurando la flessibilità nella progettazione.
- **Conversione Analogico-Digitale (ADC):** La conversione dei segnali analogici in digitali è fondamentale per permettere al microcontrollore di interagire con il mondo reale, garantendo precisione e affidabilità nelle misurazioni.

**L'importanza del Physical Computing.** La capacità di integrare input, elaborazione e output consente di creare applicazioni interattive e intelligenti, dalla domotica ai robot mobili, passando per le stazioni meteo IoT. Il segreto del successo risiede nella comprensione delle caratteristiche dei segnali e nella scelta del protocollo di comunicazione più adatto.

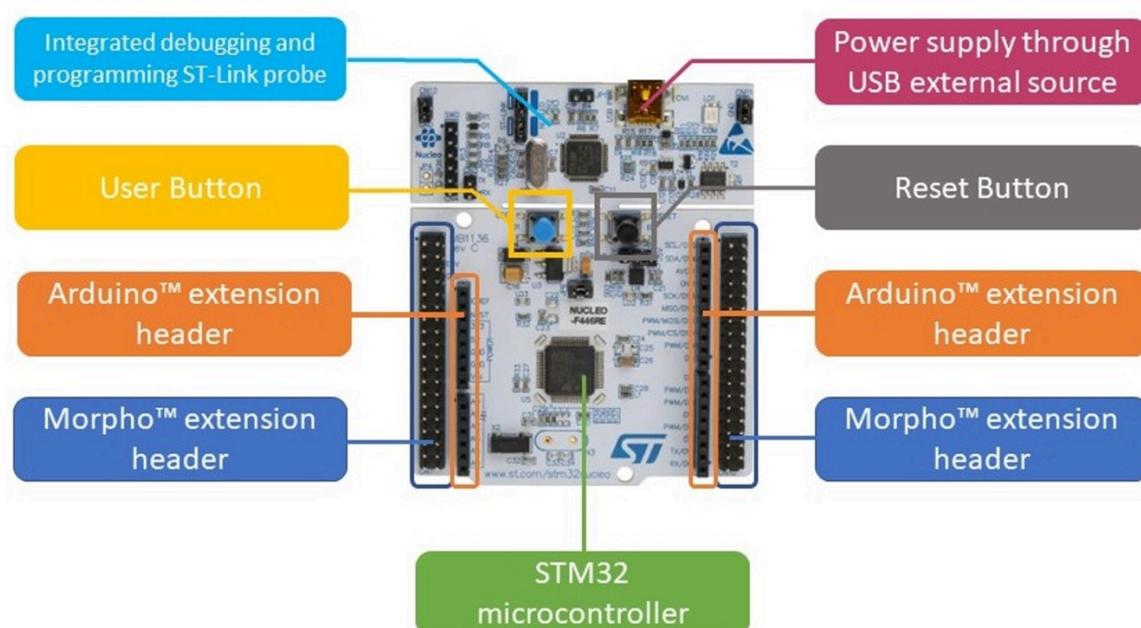
# Capitolo 2 La scheda Nucleo F401RE

La NucleoF401RE è una scheda di sviluppo progettata da STMicroelectronics, basata sul microcontrollore STM32F401RE, parte della famiglia STM32. Questa scheda è ideale per il prototipaggio e lo sviluppo di applicazioni embedded, grazie alla sua flessibilità, alle interfacce standard (Arduino e Morpho) e alla facilità di programmazione tramite ST-Link.

Lo STM32F401RE è un microcontrollore della serie STM32F4, dotato di un core ARM Cortex-M4 con unità di calcolo in virgola mobile (FPU - Floating Point Unit).

Le caratteristiche principali sono:

- **Core:** ARM Cortex-M4 a 84 MHz.
- **Memoria Flash:** 512 KB.
- **RAM:** 96 KB.
- **Porte I/O:** Fino a 50 GPIO configurabili ed interrompibili.
- **Timer:** Multipli timer avanzati e di base.
- **Interfacce:** Supporta UART, I2C, SPI e USB 2.0 OTG (On-The-Go).
- **ADC/DAC:** ADC a 12 bit.
- **Consumo energetico:** Ottimizzato per applicazioni a basso consumo.



## STM32 Microcontroller Naming Convention

Il *naming convention* per i microcontrollori STM32 codifica le loro caratteristiche. Per il microcontrollore **STM32F401RE** queste sono codificate come segue:

- **STM32**: Indica la famiglia di microcontrollori prodotta da STMicroelectronics.
- **F**: Serie del microcontrollore. La lettera "F" rappresenta i microcontrollori ad alte prestazioni.
- **4**: Serie specifica, orientata a prestazioni avanzate.
- **01**: Modello specifico all'interno della serie F4.
- **R**: Indica il tipo di package (LQFP64 per la NucleoF401RE).
- **E**: Indica la dimensione della memoria Flash (512 KB).

## Funzionalità hardware della NucleoF401RE

La scheda NucleoF401RE offre le seguenti funzionalità hardware:

- **LED Utente e Pulsante Programmabile**: LED di stato (LD2) connesso al pin PA5, può essere utilizzato per indicare stati o segnali nel programma.
- **Pulsante utente (USER)**: Il pulsante blu è connesso al pin **PC13** ed è programmabile per interagire con il firmware.
- **Pulsante di Reset**: Il pulsante nero consente di riavviare il microcontrollore senza dover scollegare e ricollegare l'alimentazione.
- **Standard dei Pin: Arduino e Morpho**:
  - **Standard Arduino Uno R3**: La scheda è compatibile con le schede di espansione Arduino, grazie alla disposizione standard dei pin. Questo semplifica l'integrazione con sensori e moduli esistenti.
  - **Connettori ST Morpho**: Oltre alla disposizione Arduino, sono disponibili pin aggiuntivi tramite i connettori Morpho, che consentono l'accesso a tutte le funzionalità del microcontrollore, come GPIO aggiuntivi, ADC e interfacce seriali.
- **ST-Link Integrato**: La scheda è dotata di un programmatore/debugger **ST-Link/V2-1** integrato, eliminando la necessità di dispositivi esterni. Questo consente:
  - Programmazione del firmware.
  - Debug tramite **SWD** (Serial Wire Debug).
  - Un'interfaccia USB virtuale per la comunicazione seriale (**VCP**).

La NucleoF401RE è una piattaforma potente e versatile per lo sviluppo embedded. La combinazione del microcontrollore **STM32F401RE** con interfacce standard come Arduino e Morpho, l'ST-Link integrato e le periferiche facilmente accessibili la rendono adatta sia per progetti di principianti sia per progetti avanzati.

## Etichette dei pin

Nel contesto dell'elettronica e dello sviluppo su microcontrollori, le etichette dei pin sono identificatori utilizzati per descrivere la funzione o lo scopo dei singoli pin su un microcontrollore, un circuito integrato (IC) o altri componenti elettronici. Queste etichette aiutano gli utenti a comprendere il ruolo del pin e a collegarlo correttamente in un circuito.

Le categorie comuni delle etichette dei pin sono:

- **Alimentazione e Massa (Power and Ground):**
  - **VCC / VDD / VIN:** Ingresso di alimentazione (tensione positiva).
  - **GND:** Connessione di massa (riferimento a 0V).
  - **3V3 / 5V:** Uscite o ingressi a tensioni specifiche.
- **Ingressi/Uscite Digitali (Digital Input/Output):**
  - **GPIO:** Pin di ingresso/uscita generici. Spesso etichettati come P0.0, P1.1, ecc., o con il numero di porta e pin.
  - **PWM:** Pin in grado di generare segnali a modulazione di larghezza di impulso.
  - **INT:** Pin di interruzione utilizzati per rilevare eventi specifici.
- **Ingressi/Uscite Analogici (Analog Input/Output):**
  - **ADC / A IN:** Pin di ingresso per il convertitore analogico-digitale.
  - **DAC / A OUT:** Pin di uscita per il convertitore digitale-analogico.
- **Comunicazione:**
  - **UART:**
    - **TX:** Linea di trasmissione.
    - **RX:** Linea di ricezione.
  - **I<sup>2</sup>C:**
    - **SCL:** Linea del clock seriale.
    - **SDA:** Linea dei dati seriali.
  - **SPI:**
    - **SCK:** Clock seriale.
    - **MOSI:** Master Out Slave In.
    - **MISO:** Master In Slave Out.
    - **CS / SS:** Chip Select / Slave Select.
  - **CAN:**
    - **CANH:** CAN High.
    - **CANL:** CAN Low.
- **Funzioni Speciali (Special Functions):**
  - **RESET / RST:** Pin utilizzato per resettare il dispositivo.
  - **BOOT:** Pin utilizzato per configurare la modalità di avvio.
  - **CLK:** Pin di ingresso/uscita per il clock.
  - **EN:** Pin di abilitazione per accendere o spegnere un dispositivo.
  - **LED / IND:** Pin per LED indicativi.

I produttori di microcontrollori generalmente etichettano i pin in base a:

- **Raggruppamento funzionale:** I pin possono avere funzioni multiple (ad esempio, un pin GPIO che funge anche da linea SDA per l'I<sup>2</sup>C).
- **Mappatura dei pin:** Definita nel datasheet del microcontrollore, spesso con diagrammi o tabelle che mostrano le configurazioni possibili.

Esempio:

Un pin potrebbe essere etichettato come **P1.0/TXD/PWM1**, indicando che può funzionare come:

- Ingresso/Uscita generale (GPIO) **P1.0**.
- Trasmissione UART (**TXD**).
- Uscita PWM (**PWM1**).

Quando si lavora con un microcontrollore, fare sempre riferimento al datasheet o al diagramma dei pin per una descrizione accurata delle etichette e delle configurazioni dei pin.

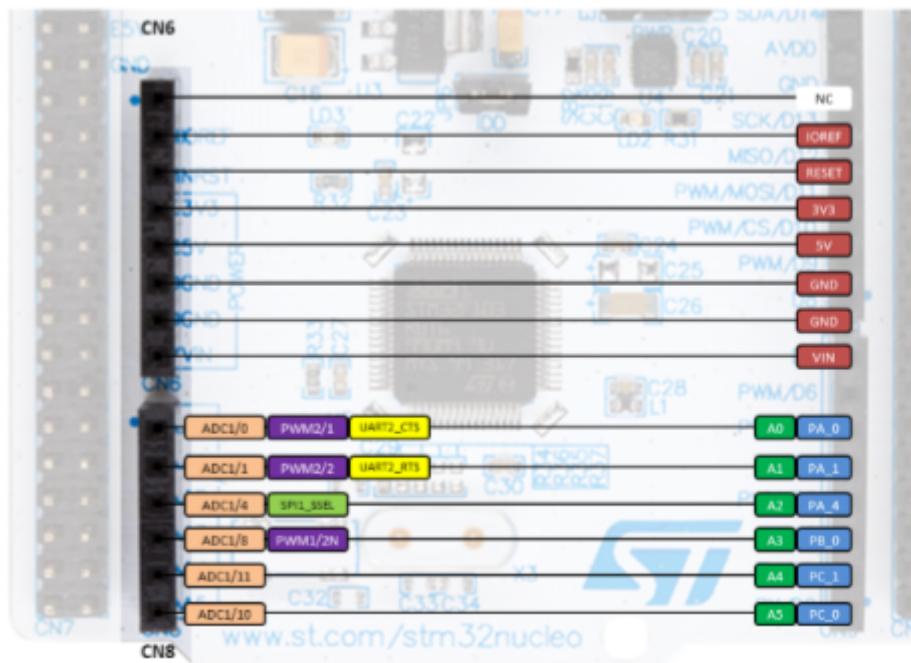
#### Labels usable in code

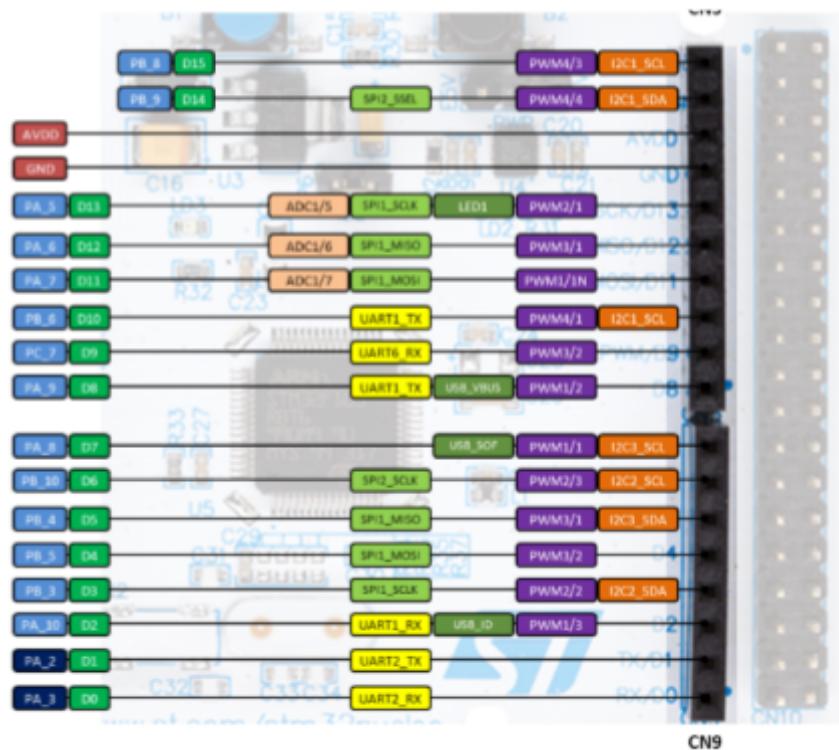
<b>PX_Y</b>	MCU pin without conflict	<b>XXX</b>	Arduino connector names (A0, D1, ...)
<b>PX_Y</b>	MCU pin connected to other components	<b>XXX</b>	LEDs and Buttons (LED_1, USER_BUTTON, ...)

#### Labels not usable in code (for information only)

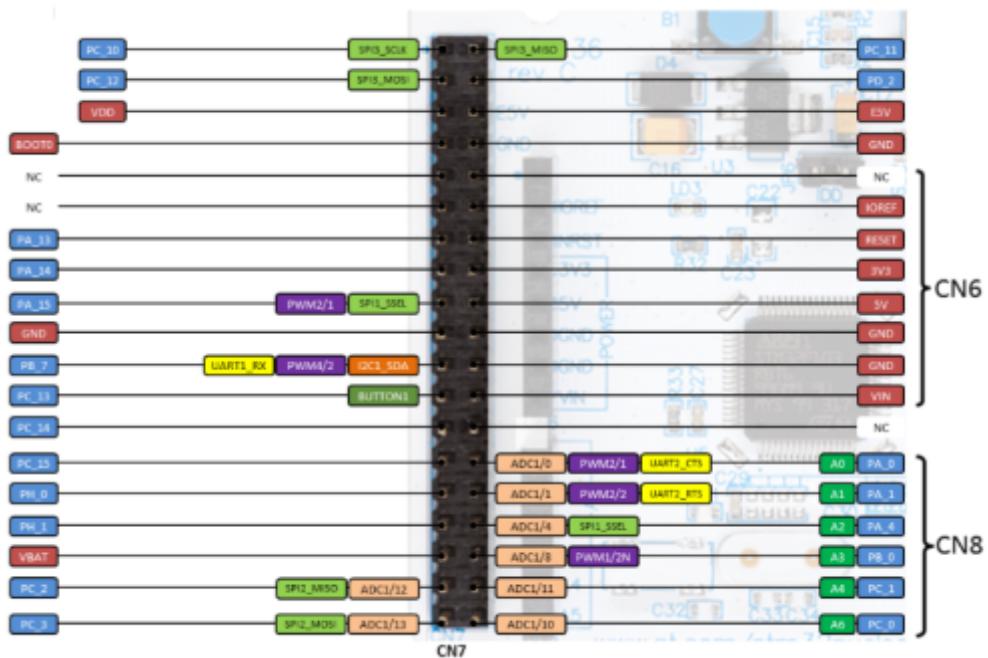
<b>XXX</b>	Serial pins (USART/UART)	<b>XXX</b>	AnalogIn (ADC) and AnalogOut pins (DAC)
<b>XXX</b>	SPI pins	<b>XXX</b>	CAN pins
<b>XXX</b>	I2C pins		
<b>XXX</b>	PWMOut pins (TIMER n/c[N]) n = Timer number c = Channel N = Inverted channel	<b>XXX</b>	Power and control pins (3V3, GND, RESET, ...)

#### Arduino-compatible headers





Morpho headers. Questi headers danno accesso a tutti i pin del STM32.





## CubeIDE

Un **Integrated Development Environment (IDE)** è un ambiente di sviluppo software integrato progettato per semplificare la scrittura, il debug e la compilazione del codice. Un IDE riunisce in un'unica interfaccia grafica tutti gli strumenti essenziali per lo sviluppo, inclusi:

- **Editor di codice sorgente:** Per scrivere e modificare il codice.
- **Compilatore/Linker:** Per tradurre il codice in un linguaggio comprensibile al microcontrollore.
- **Debugger:** Per individuare e correggere errori nel codice.
- **Simulatori o strumenti di programmazione:** Per testare il codice direttamente su hardware reale.

Utilizzare un IDE consente agli sviluppatori di risparmiare tempo e ridurre gli errori, poiché elimina la necessità di gestire strumenti separati.

**STM32CubeIDE** è l'ambiente di sviluppo ufficiale di **STMicroelectronics** progettato per la programmazione e il debug di microcontrollori STM32. Basato su Eclipse e integrato con GCC (GNU Compiler Collection), STM32CubeIDE combina strumenti avanzati per progettare e implementare applicazioni embedded.

Le caratteristiche principali di STM32CubeIDE sono:

- **Integrazione con STM32CubeMX:** STM32CubeIDE include **STM32CubeMX**, un potente configuratore grafico che permette di configurare i pin, le periferiche e i clock

del microcontrollore tramite un'interfaccia visiva. Genera automaticamente il codice di inizializzazione, riducendo il tempo richiesto per configurare il microcontrollore.

- **Supporto per Debug Avanzato:** Compatibile con i debugger hardware come **ST-Link** e supporta funzionalità avanzate come breakpoint, watchpoints e analisi delle variabili in tempo reale.
- **Supporto Multi-Toolchain:** Utilizza GCC come compilatore di default ma supporta altre toolchain, offrendo flessibilità nello sviluppo.
- **Gestione del Firmware:** Consente di scaricare e aggiornare i pacchetti firmware STM32 direttamente dall'interfaccia, assicurando che i progetti siano sempre aggiornati.
- **Cross-Platform:** Disponibile per Windows, macOS e Linux, rendendolo accessibile a una vasta comunità di sviluppatori.

STM32CubeIDE è ideale per sviluppatori di tutti i livelli, grazie alla sua interfaccia intuitiva e alla profonda integrazione con la famiglia di microcontrollori STM32. È particolarmente utile per chi cerca:

- Una piattaforma **tutto in uno** per la programmazione embedded.
- Strumenti potenti per progetti sia semplici che complessi.
- Supporto nativo per una vasta gamma di schede STM32.

## Installazione

### Requisiti di Sistema

Prima di iniziare, verifica che il tuo computer soddisfi i requisiti minimi:

- **Sistema operativo:**
  - Windows 10 o versioni successive (64-bit).
  - macOS 10.15 o successivo.
  - Distribuzioni Linux come Ubuntu 18.04/20.04/22.04 (64-bit).
- **RAM:** Almeno 4 GB (consigliati 8 GB o più).
- **Spazio su disco:** Circa 1.5 GB per l'installazione.
- **Java:** Non è richiesto, STM32CubeIDE include il runtime Java necessario.

### Scaricare STM32CubeIDE

- Vai al sito ufficiale di STMicroelectronics:  
<https://www.st.com/en/development-tools/stm32cubeide.html>
- Seleziona la tua piattaforma (Windows, macOS, Linux) e clicca su **Download**.

- **Registrazione (opzionale):** Potrebbe essere richiesto di creare un account o effettuare il login per scaricare il software.

## Installazione su Windows

- **Esegui il file di installazione** scaricato (`STM32CubeIDE-Win64-<version>.exe`).
- Segui i passaggi dell'installazione guidata:
  - Accetta i termini e condizioni.
  - Scegli la directory di installazione (es.: `C:\STMicroelectronics\STM32CubeIDE`).
- Seleziona le componenti aggiuntive (opzionali):
  - **Supporto per OpenOCD**.
  - **Driver ST-Link**.
- Clicca su **Installa** e attendi il completamento.

## Installazione su macOS

- **Esegui il pacchetto DMG** scaricato (`STM32CubeIDE-<version>.dmg`).
- Trascina l'app STM32CubeIDE nella cartella **Applicazioni**.
- Se richiesto, consenti l'esecuzione di app scaricate da sviluppatori identificati:
  - Vai in **Preferenze di Sistema > Sicurezza e Privacy > Generali**.
  - Clicca su **Consenti**.

## Installazione su Linux

- **Rendi eseguibile il file scaricato:**  
`chmod +x STM32CubeIDE-<version>.linux64.run`
- **Esegui l'installazione:**  
`./STM32CubeIDE-<version>.linux64.run`
- Segui le istruzioni nel terminale per completare l'installazione.
- **Aggiungi i driver USB udev rules** per ST-Link:
  - Scarica le regole da: [ST-Link udev rules](#).
  - Copia il file `.rules` nella directory `/etc/udev/rules.d/`.
- Ricarica le regole:
  - `sudo udevadm control --reload-rules`

## Configurazione di STM32CubeIDE

- **Avvia STM32CubeIDE:**
  - Windows: Clicca sull'icona del menu Start.
- macOS/Linux: Trova l'app STM32CubeIDE nelle applicazioni o esegui il comando:  
`./STM32CubeIDE`

- **Imposta la directory di lavoro (workspace):**
  - Scegli una cartella per i tuoi progetti (es.: C:\WS\_CoreOfEmbeddedSystems).

## Introduzione a STM32CubeMX

**STM32CubeMX** è un potente strumento software fornito da STMicroelectronics per configurare, inizializzare e generare codice per i microcontrollori STM32. Può essere utilizzato sia come applicazione stand-alone sia come modulo integrato nell'ambiente di sviluppo **STM32CubeIDE**.

Le principali funzionalità sono:

- **Configurazione grafica del microcontrollore:** Consente di configurare i pin e le periferiche attraverso un'interfaccia grafica user-friendly.
- **Configurazione del clock:** Facilita l'impostazione dei clock del sistema (es. HSE, LSE, PLL) con una rappresentazione visiva chiara.
- **Generazione automatica del codice:** Produce il codice C per inizializzare periferiche e configurazioni.
- **Supporto per middleware:** Configura e genera codice per librerie come FreeRTOS, FatFS, USB e altre.
- **Simulazione di consumo energetico:** Include strumenti per stimare il consumo energetico del microcontrollore in diverse configurazioni.

STM32CubeMX è disponibile in due modalità:

- **Tool Stand-Alone:** Scaricabile gratuitamente dal sito ufficiale di STMicroelectronics. Ideale per chi vuole generare il codice e utilizzarlo in ambienti di sviluppo differenti (es. Keil, IAR).
- **Modulo integrato in STM32CubeIDE:** In questa modalità, STM32CubeMX è perfettamente integrato nell'ambiente di sviluppo, permettendo di passare rapidamente dalla configurazione hardware alla scrittura del codice e al debug.

# Capitolo 3 GPIO output

La **GPIO** (General Purpose Input/Output) è una delle periferiche più importanti di un microcontrollore. I pin GPIO possono essere configurati come input o output digitali, rendendoli estremamente versatili per interfacciarsi con il mondo esterno. In questo progetto introduttivo, impareremo a configurare un pin GPIO come output digitale per far lampeggiare il LED verde della scheda NucleoF401RE utilizzando STM32CubeIDE.

## Obiettivi del Progetto

- Comprendere il funzionamento di un pin GPIO come output digitale.
- Configurare un progetto base in STM32CubeIDE.
- Controllare il LED verde della NucleoF401RE utilizzando codice C.
- Introdurre il concetto di ritardi temporali per generare un lampeggio visibile.

## Teoria di Base: Cos'è un GPIO?

I GPIO sono pin configurabili che possono funzionare in modalità:

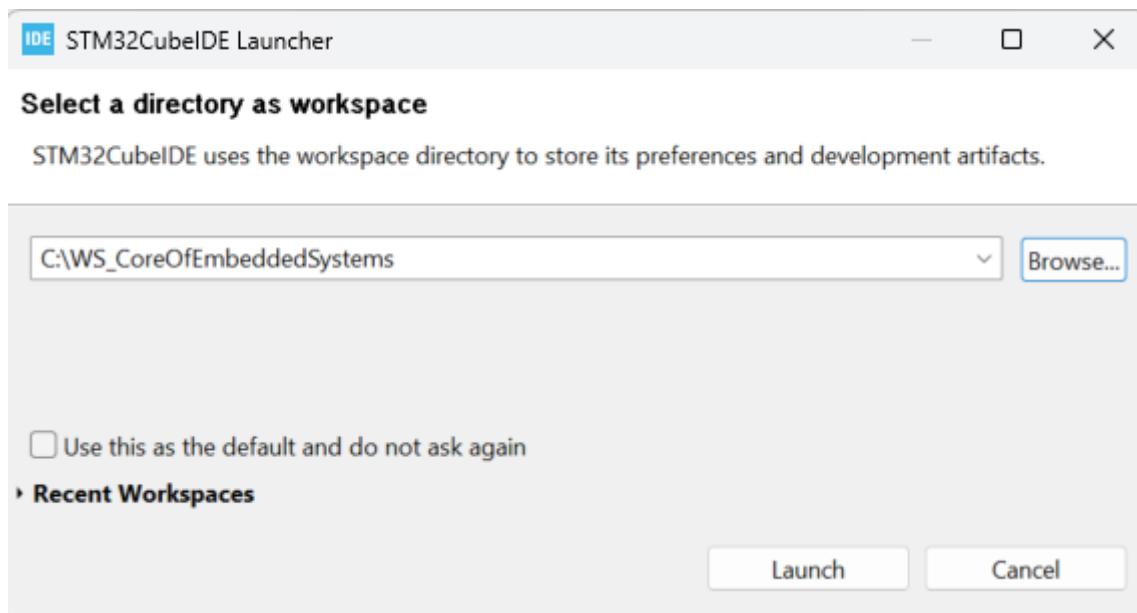
- **Input digitale:** Per leggere lo stato logico (alto o basso) di un segnale esterno.
- **Output digitale:** Per generare uno stato logico (alto o basso) e controllare dispositivi esterni come LED o relè.

Quando un pin GPIO è configurato come output digitale:

- Uno stato **alto (HIGH)** corrisponde tipicamente a una tensione positiva (es. 3.3V).
- Uno stato **basso (LOW)** corrisponde a una tensione vicina a 0V.

## Configurare il Progetto

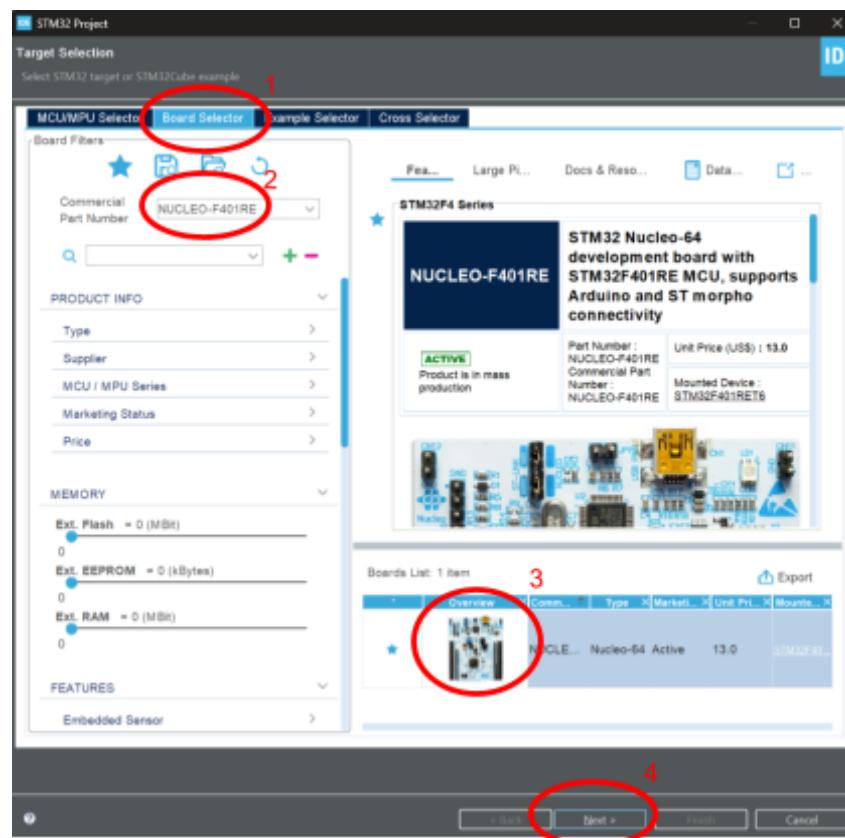
Apri STM32CubeIDE e seleziona il workspace creato in precedenza



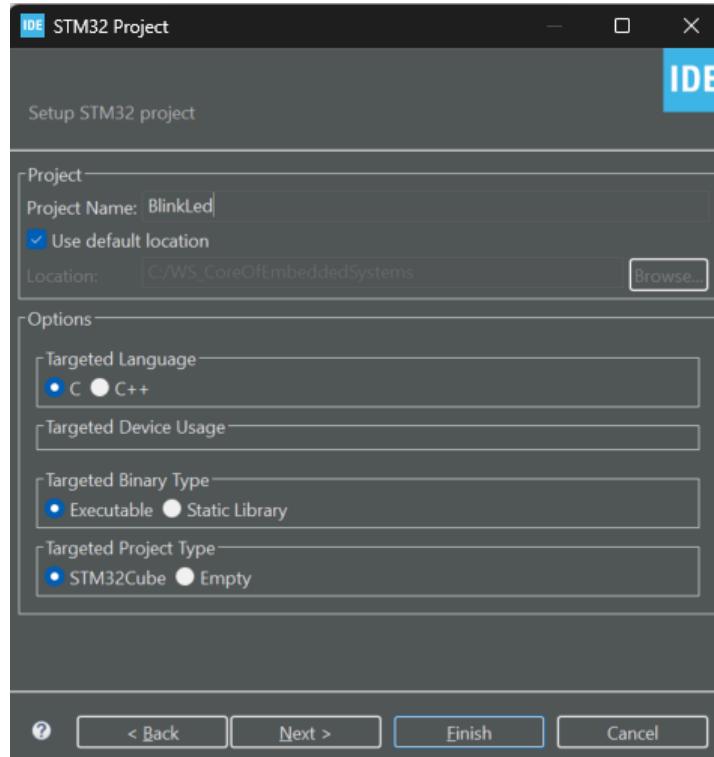
Crea un nuovo progetto in *File/New/STM32 Project*. In questa fase CubeIDE, se necessario, eseguirà un aggiornamento

Seleziona *Board Selector*, la scheda **Nucleo-F401RE** nel campo *Commercial Part Number*, e come *item* nella *Board List*. Infine il pulsante *Next*.

In questo modo abbiamo scelto la scheda Target del progetto.

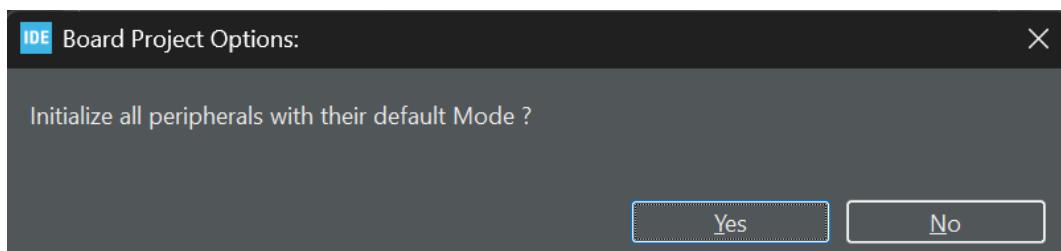


Si aprirà una finestra per il *Setup STM32 project*. Inseriamo il nome del progetto *BlinkLed* lasciando tutto il resto invariato.



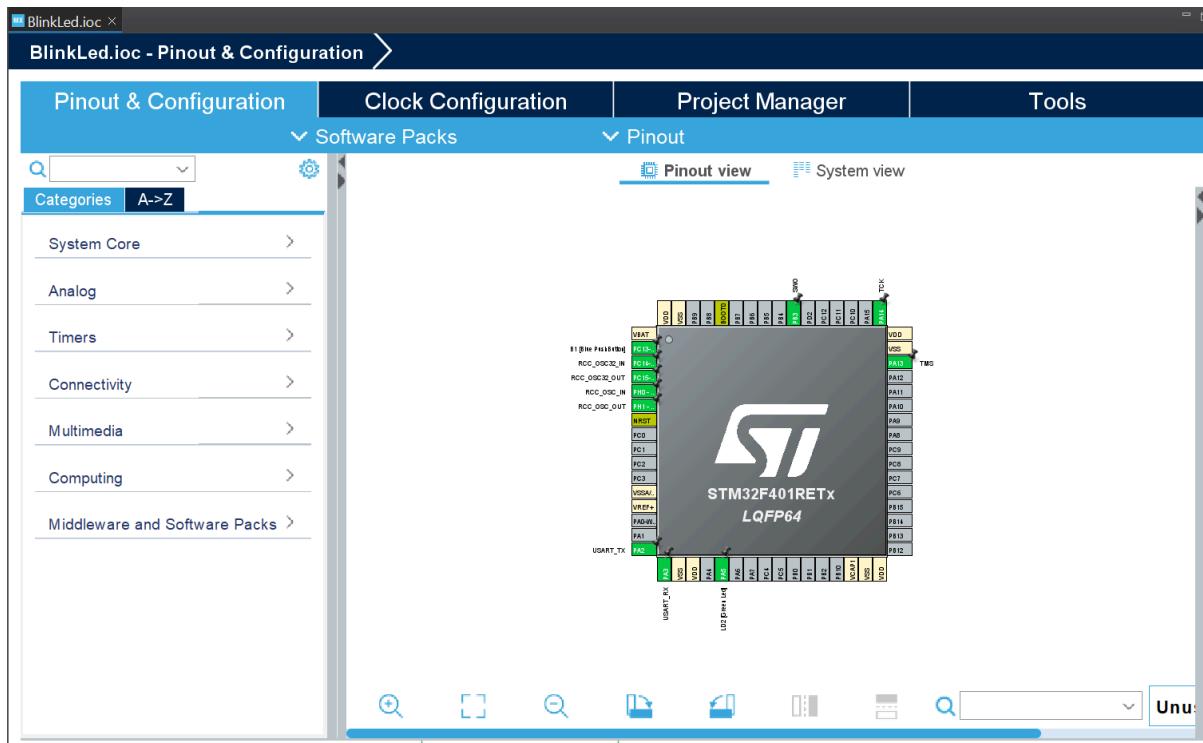
Non dovendo definire altro, possiamo chiudere il setup cliccando il pulsante *Finish*.

Lasciamo che il software inizializzi le periferiche nella modalità di default con un clic su *Yes*.



Cliccando *Yes* si avvia la generazione del progetto. Al completamento della generazione, al centro dell'ambiente di sviluppo, si aprirà un'interfaccia grafica che consente la configurazione del microcontrollore, del Pinout, del Clock e del progetto. Sono inoltre disponibili degli strumenti che riguardano il consumo ed il design.

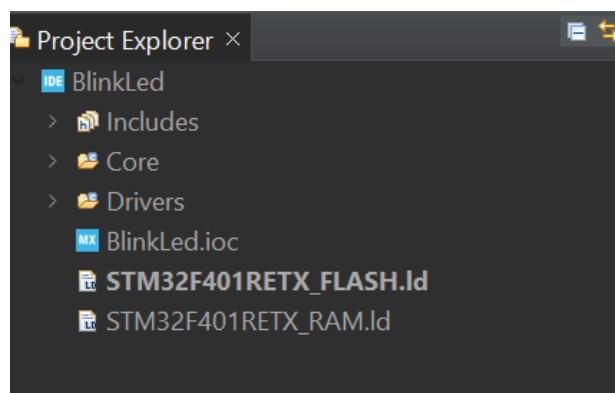
In questo manuale ci occuperemo esclusivamente della configurazione del Pinout, lasciando al lettore approfondimenti successivi.



Le configurazioni che faremo saranno sempre modificabili essendo queste salvate nel file *BlinkLed.ioc*.

Prima di procedere con la configurazione dei pin di GPIO, esaminiamo la struttura di un progetto di CubieIDE. Questa è la stessa per tutti i progetti.

Sulla sinistra troviamo il *Project Explorer*



Nel contesto di un tipico progetto STM32CubieIDE, le cartelle *includes*, *Core*, e *Drivers* sono fondamentali per organizzare il codice sorgente e le risorse del progetto.

La Cartella ***Includes*** è un alias o una scorciatoia che raccoglie i percorsi delle intestazioni (file .h) utilizzate nel progetto. Non contiene file direttamente ma serve per organizzare l'ambiente di sviluppo.

Il Contenuto tipico: sono collegamenti a file .h da altre directory, come quelli generati automaticamente da STM32CubeMX o inclusi nella cartella *Drivers*. Rende più semplice

includere file di intestazione nel codice senza dover specificare percorsi complessi. Ad esempio, si può scrivere `#include "stm32f4xx_hal.h"` invece di specificare il percorso completo.

La cartella **Core** contiene i file principali del progetto, sia generati automaticamente da STM32CubeMX che scritti dall'utente. Questa cartella è cruciale per il funzionamento dell'applicazione.

Le Sottocartelle principali sono:

- **Inc/**: contiene i file di intestazione (.h) specifici del progetto come ad esempio il file `main.h`, che include definizioni globali, macro, e dichiarazioni di variabili o funzioni.
- **Src/**: contiene i File sorgenti (.c) che includono il codice principale del progetto, come ad esempio il File `main.c`, dove risiede il punto di ingresso del programma e la logica applicativa. Altri file possono includere implementazioni specifiche, come callback o gestori di interrupt.
- **Startup/**: contiene il file di startup del microcontrollore (`startup_stm32xxx.s`), scritto in assembly, che gestisce l'inizializzazione del sistema (es. stack, vettori di interrupt).

La cartella **Core** è dove si scrive e si organizza il codice personalizzato per l'applicazione.

La cartella **Drivers** contiene le librerie di basso livello (LL - Low Level) e di astrazione hardware (HAL - Hardware Abstraction Layer) fornite da STM32Cube per l'interazione con le periferiche del microcontrollore.

Le Sottocartelle principali sono:

- **CMSIS/**:
  - **CMSIS (Cortex Microcontroller Software Interface Standard)**:  
Librerie standard fornite da ARM per l'accesso a registri di basso livello e per operazioni specifiche del core Cortex.
  - File tipici: `core_cm4.h`, `system_stm32xxx.c`, `stm32f4xx_it.c`.
- **STM32XXxx\_HAL\_Driver/**:
  - Contiene i file driver HAL (Hardware Abstraction Layer) per gestire periferiche come GPIO, UART, SPI, ADC, ecc.
  - File tipici: `stm32f4xx_hal_gpio.c`, `stm32f4xx_hal_rcc.c`, ecc.
- **BSP/ (opzionale)**:
  - **Board Support Package**: Contiene driver specifici per le schede di sviluppo STM32 (es. Nucleo, Discovery).

I file in questa cartella vengono utilizzati per semplificare la configurazione e il controllo delle periferiche hardware.

Questa organizzazione permette di mantenere il progetto modulare, scalabile e leggibile, favorendo un flusso di lavoro chiaro e una separazione tra configurazioni hardware, librerie e codice applicativo.

## Compilare il progetto

La compilazione è il processo mediante il quale il codice sorgente scritto in un linguaggio di programmazione (ad esempio, C o C++) viene tradotto in un linguaggio macchina comprensibile dal computer o microcontrollore. Questo linguaggio macchina è costituito da istruzioni binarie che il processore può eseguire direttamente.

La compilazione si compone di più fasi, che possono variare leggermente in base al compilatore utilizzato, ma generalmente includono:

- **Preprocessing (Pre-elaborazione):** Il preprocessore analizza il codice sorgente prima della compilazione vera e propria.
  - Esegue operazioni come:
    - i. Espansione delle macro.
    - ii. Inclusione dei file di intestazione (`#include`).
    - iii. Gestione delle direttive condizionali (`#ifdef`, `#endif`).
  - Il risultato è un file sorgente preprocessato.
- **Parsing e Analisi Sintattica:** Il compilatore analizza il codice sorgente per verificarne la correttezza sintattica (conformità alle regole del linguaggio).
- **Traduzione in Codice Intermedio:** Il codice sorgente viene tradotto in una rappresentazione intermedia (ad esempio, `assembly`) per facilitarne l'ottimizzazione e la successiva traduzione.
- **Ottimizzazione:** Il compilatore applica tecniche per rendere il codice più efficiente in termini di velocità o consumo di memoria.
- **Generazione del Codice Oggetto:** Il codice intermedio ottimizzato viene tradotto in **codice oggetto** (file `.o` o `.obj`), che contiene istruzioni macchina ma è ancora incompleto.
- **Linking (Collegamento):** I file oggetto vengono combinati con eventuali librerie esterne per produrre un file eseguibile completo (ad esempio, un file `.elf`, `.bin` o `.exe`).

I Compilatori possono essere:

- **Compilatori nativi:** Creano codice macchina eseguibile direttamente sull'hardware del sistema (es. GCC per ARM o x86).
- **Compilatori cross-platform (Cross-compiler):** Generano codice eseguibile per un'architettura diversa rispetto a quella del sistema in uso (es. ARM GCC per microcontrollori STM32).

In STM32CubeIDE, la compilazione converte il codice sorgente scritto in C/C++ (ad esempio, main.c) in un file binario (.bin o .elf) che può essere caricato ed eseguito su un microcontrollore STM32.

Il progetto generato non fa nulla, ma è già compilabile ed è possibile scaricare il file binario nella memoria del microcontrollore.

Vediamo come si procede per l'esecuzione di queste due operazioni.

Per compilare è sufficiente cliccare l'icona del martello.



Il processo di compilazione genera dei messaggi nella *Console* che si conclude con la stampa del numero di *errori* e di *warning*.

A screenshot of the STM32CubeIDE interface focusing on the 'Console' tab. The tab bar at the top includes 'Problems', 'Tasks', 'Console X', and 'Properties'. The console window displays the following text:

```
CDT Build Console [BlinkLed]
text      data      bss      dec      hex filename
 7652        12     1644    9308    245c BlinkLed.elf
Finished building: default.size.stdout

Finished building: BlinkLed.list

18:53:07 Build Finished. 0 errors, 0 warnings. (took 986ms)
```

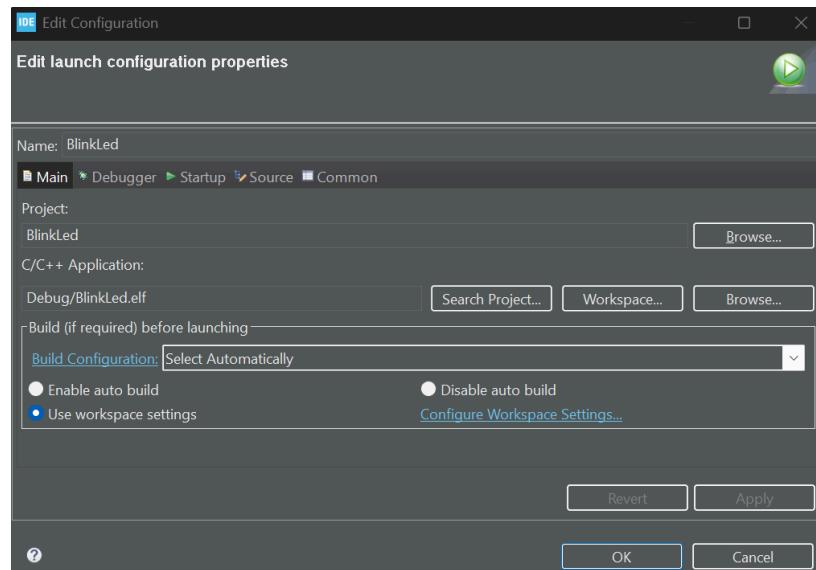
Se il processo di generazione del progetto ha funzionato correttamente, la compilazione si conclude correttamente senza errori.

# Programmazione del microcontrollore

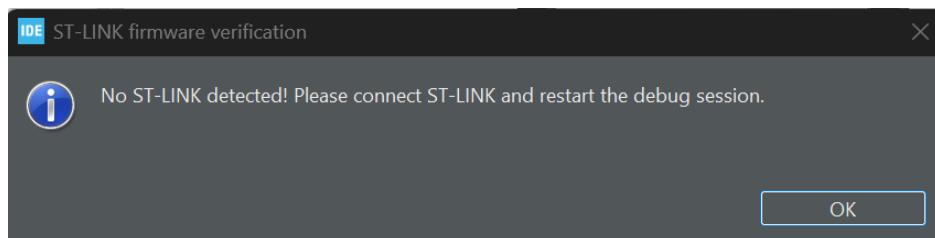
Per programmare la scheda Nucleo con il codice appena compilato, è sufficiente cliccare l'icona del triangolo



La prima volta è necessario accettare la configurazione di default con un clic su **OK**



Se la scheda Nucleo non è collegata al PC, un popup ci riferirà dell'assenza di ST-Link.

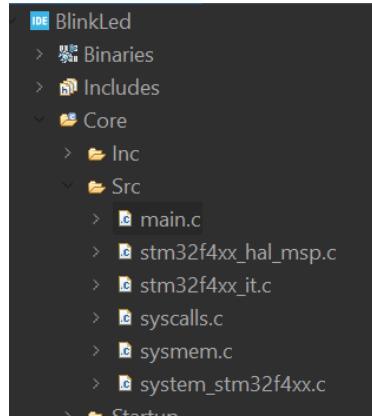


In tal caso sarà necessario collegare la scheda Nucleo al PC e ripetere l'operazione.

## Il file main.c

Il file `main.c`, generato da CubeMX per una scheda NucleoF401RE, è il file principale di un progetto basato su HAL (Hardware Abstraction Layer) per il microcontrollore STM32F401RE. Contiene la configurazione iniziale e il ciclo principale del firmware.

Questo file si trova in `Core/Src`



Il file include vari header necessari per il funzionamento del codice:

- `stm32f4xx_hal.h`: Fornisce le definizioni e le API per la libreria HAL.
- Header specifici per periferiche configurate (es. `gpio.h`, `usart.h`, `tim.h`).

Al suo interno sono presenti le dichiarazioni di funzioni di configurazione come:

- `void SystemClock_Config(void)`; che configura il clock del sistema.
- `void MX_GPIO_Init(void)`; che configura i pin GPIO.
- Altre funzioni specifiche per configurare le periferiche (es. UART, Timer).

La funzione principale `main()` è il punto di ingresso del programma. Questa invoca:

- l'Inizializzazione della HAL, `HAL_Init()`;
- la configurazione del Clock di sistema nella funzione `SystemClock_Config()`.
- l'Inizializzazione delle Periferiche. Chiama le funzioni di inizializzazione generate da CubeMX, come `MX_GPIO_Init()` e `MX_USART2_UART_Init()`;
- inoltre include il Ciclo Principale (while):

```
while (1)
{
    // Funzioni personalizzate
}
```

CubeMX include commenti per definire dove scrivere il proprio codice:

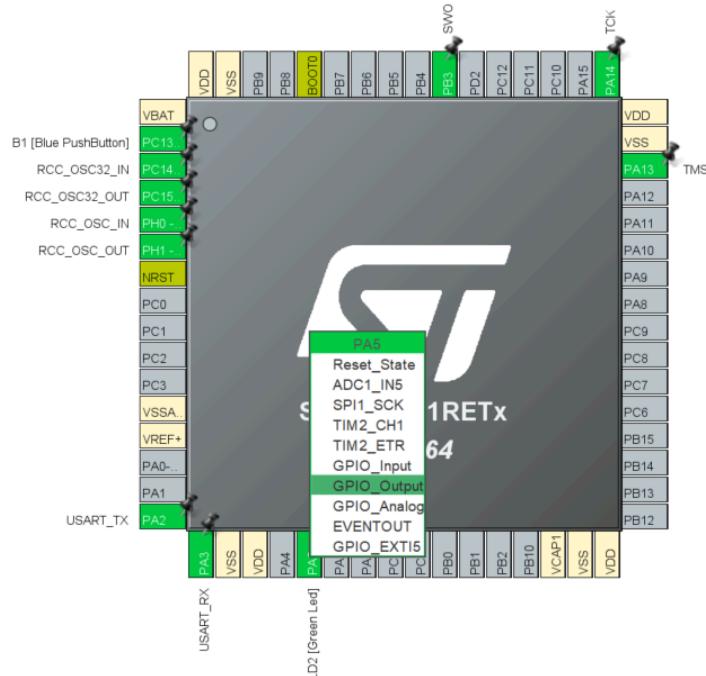
- `/* USER CODE BEGIN 0 */` e `/* USER CODE END 0 */` per definire variabili globali.
- `/* USER CODE BEGIN WHILE */` e `/* USER CODE END WHILE */` per inserire codice personalizzato nel ciclo principale.

Sostanzialmente, il file `main.c` è un punto di partenza generato automaticamente, strutturato per essere estendibile. Gli sviluppatori aggiungono il proprio codice nelle sezioni designate senza alterare le parti generate automaticamente, facilitando la rigenerazione in CubeMX senza perdere modifiche. Il file include i file header necessari ed incorpora tutte le funzioni di inizializzazione che vengono chiamate nella funzione `main()`.

## Configurazione del Pin

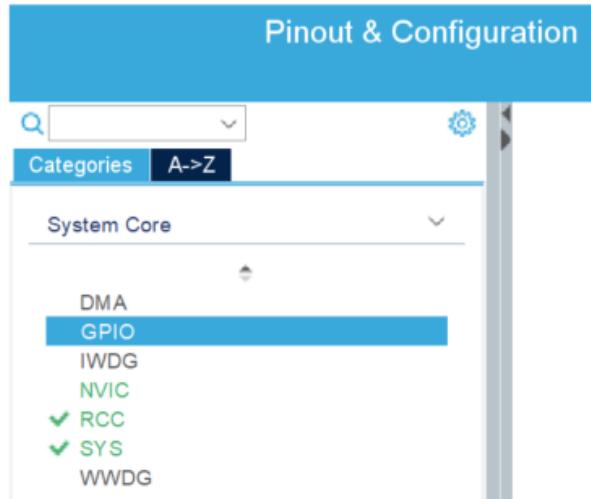
Nella vista del microcontrollore identifica il pin connesso al LED verde. Per la Nucleo-F401RE, il LED verde è connesso al pin PA5.

CubeMX dovrebbe aver configurato il pin come un GPIO output. Se il pin risulta di colore verde è già configurato. Ad ogni modo possiamo procedere come segue per analizzare la configurazione:



Fai clic su PA5 e seleziona **GPIO\_Output** tra tutte le possibili *alternate functions* disponibili per il pin.

In *System Core* seleziona GPIO



In questo modo saranno visualizzati tutti i pin configurati come GPIO

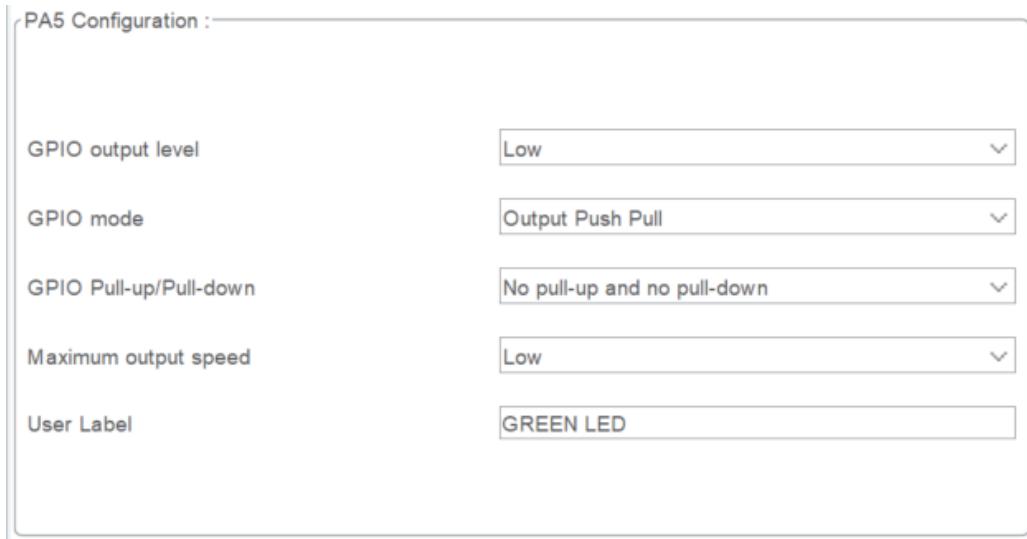
GPIO Mode and Configuration							
Configuration							
Group By Peripherals							
GPIO	RCC	SYS	USART	NVIC			
<input checked="" type="checkbox"/>							
Search Signals	Search (Ctrl+F)				<input type="checkbox"/> Show only Modified Pins		
Pin Name	Signal on Pin	GPIO output	GPIO mode	GPIO Pull	Maximum ...	User Label	Modified
PA5	n/a	Low	Output Pu...	No pull-up ...	Low		<input type="checkbox"/>
PC13-ANT...	n/a	n/a	External I...	No pull-up ...	n/a	B1 [Blue P...	<input checked="" type="checkbox"/>

Il pin **PA5** (LED verde) nella lista va configurato come segue:

- **Mode:** Output Mode
  - i. In un GPIO configurato in modalità **push-pull**, il pin può pilotare attivamente sia lo stato logico alto (collegandosi alla tensione di alimentazione) sia lo stato logico basso (collegandosi a GND). In altre parole, il microcontrollore fornisce internamente la corrente sia quando il pin è a livello alto sia quando è a livello basso. Questa configurazione è tipicamente quella predefinita per uscite digitali standard, perché garantisce transizioni veloci e non necessita di componenti esterni.
  - ii. In un GPIO configurato in modalità **open-drain**, invece, quando il transistor interno è “aperto” il pin risulta scollegato (alta impedenza) e non fornisce alcuna tensione, mentre quando il transistor è “chiuso” il pin viene collegato a massa (GND). Per poter avere un livello alto su un’uscita open-drain, serve obbligatoriamente una resistenza di pull-up esterna (o interna, se disponibile nel microcontrollore).

- **Pull-up/Pull-down:** No Pull (di solito non è necessario, ma dipende dal design specifico)
- **Speed:** Low, Medium, High (scegli in base alle tue esigenze; per un LED è generalmente sufficiente **Low**)
- **Output Type:** Push-Pull

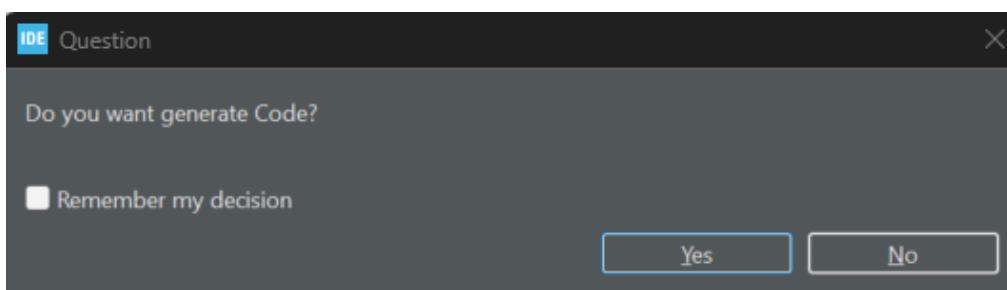
Con un clic sul pin PA5 è possibile aprire la finestra di configurazione.



Per questo progetto vanno bene i valori di configurazione di default. Possiamo definire un'etichetta *User Label* che aiuti ad individuare la funzionalità del pin. Questo è possibile farlo per tutti i pin, a prescindere dalla funzionalità.

Con queste operazioni, il pin PA5 è stato configurato come pin di GPIO output e possiamo iniziare ad utilizzarlo.

Salvando il progetto, CubeMX ci chiede se vogliamo generare il codice.



Accettiamo la generazione con un clic su Yes. Spuntando *Remember my decision*, la generazione del codice sarà eseguita ad ogni salvataggio.

La configurazione del pin PA5, essendo stato definito come un pio GPIO\_Output sarà presente nella funzione `static void MX_GPIO_Init(void);`

## Aspetti del Linguaggio C

- **static**: Limita la visibilità della funzione al file corrente, prevenendo conflitti di nome con altre funzioni simili in progetti complessi.
- **void**: Specifica che la funzione non accetta parametri e non restituisce un valore.

Esaminiamo il codice della funzione di inizializzazione generata da CubeMX.

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

- Crea una variabile della struttura **GPIO\_InitTypeDef**, utilizzata per configurare i parametri dei pin GPIO, come modalità, velocità, tipo di output, ecc.
- Inizializza la struttura a zero per garantire valori predefiniti.

```
__HAL_RCC_GPIOC_CLK_ENABLE();  
__HAL_RCC_GPIOH_CLK_ENABLE();  
__HAL_RCC_GPIOA_CLK_ENABLE();  
__HAL_RCC_GPIOB_CLK_ENABLE();
```

- Queste macro abilitano il clock per i diversi gruppi di GPIO (C, H, A, B).
- Senza l'abilitazione del clock, i GPIO non funzioneranno.

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

- Imposta il livello iniziale del pin **GREEN\_LED** su **RESET** (LOW).
- **GREEN\_LED\_GPIO\_Port** e **GREEN\_LED\_Pin** sono macro definite automaticamente da STM32CubeMX per identificare la porta e il pin associati al 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:** Specifica il pin del LED verde.
- **Mode:** Configura il pin come **output push-pull** (OUTPUT\_PP).
- **Pull:** Nessuna resistenza pull-up o pull-down attivata.
- **Speed:** Imposta la velocità del pin su **LOW** (sufficiente per un LED).
- **HAL\_GPIO\_Init:** Inizializza il pin con i parametri definiti.

## Lampeggiare il LED

Avendo completato la configurazione del pin PA5 a cui è collegato il LED di colore verde, possiamo procedere con l'implementazione della logica.

Per lampeggiare un LED, alterniamo lo stato del pin GPIO tra alto (HIGH) e basso (LOW).

Un ritardo («delay») è necessario tra le transizioni per rendere il lampeggio visibile.

Con questa logica dobbiamo seguire i seguenti passi:

- Scrivi **1** sul registro per accendere il LED.
- Attendi un intervallo di tempo (es. 500 ms).
- Scrivi **0** sul registro per spegnere il LED.
- Ripeti.

Per l'implementazione utilizziamo le funzioni già disponibili del HAL (Hardware Abstraction Layer)

Per accendere o spegnere il led, utilizziamo la funzione di scrittura di un pin. Questa funzione prende in ingresso la *porta* a cui appartiene il pin, il pin stesso e lo stato che si vuole scrivere.

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

Quando assegnamo un nome al pin in fase di configurazione, CubeMX genera una macro per la porta che include il pin ed una macro per il pin. In questo caso, l'etichetta assegnata al pin PA5 è GREEN\_LED, e CubeMX ha generato la macro GREEN\_LED\_GPIO\_port che individua la porta e GREEN\_LED\_Pin che individua il pin.

Le due macro predefinite, GPIO\_PIN\_SET e GPIO\_PIN\_RESET rispettivamente alzano ed abbassano l'uscita di un pin digitale.

Di seguito il codice da inserire nel loop nella sezione USER CODE.

```
/* 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 */
```

Per inserire il ritardo tra l'accensione e lo spegnimento e viceversa, utilizziamo un'altra funzione del HAL.

La funzione void HAL\_Delay(uint32\_t delay) prende in ingresso il tempo di attesa espresso in millisecondi rappresentati da un intero a 32 bit.

Il lampeggio del led si può ottenere anche utilizzando un'altra funzione del HAL.

La funzione void HAL\_GPIO\_TogglePin(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin) modifica lo stato del pin digitale di output. Se lo stato attuale è Alto, quello successivo sarà basso e viceversa.

Di seguito il codice da inserire nel loop nella sezione USER CODE.

```
/* 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 */
}
```

Compilare e scaricare il programma sulla scheda Nucleo.

# Capitolo 4 La comunicazione seriale

In questo progetto impareremo a configurare la periferica USART2 del microcontrollore STM32F401RE in modalità comunicazione seriale asincrona per inviare e ricevere dati da un terminale PC. Useremo un terminale (es. PUTTY o serial monitor generico) per trasmettere comandi testuali al microcontrollore e, in risposta, far lampeggiare o spegnere il LED verde (LD2) sulla scheda Nucleo.

## Obiettivi del Progetto

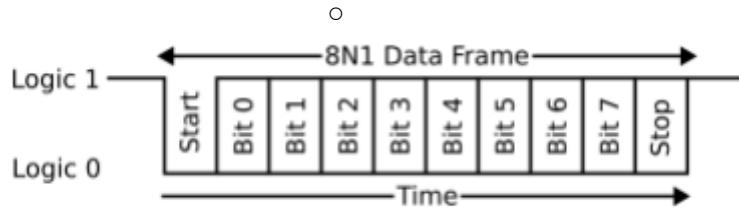
- Comprendere il funzionamento di una porta seriale UART/USART in modalità asincrona.
- Configurare un progetto base in STM32CubeIDE con USART2 (PA2=Tx, PA3=Rx).
- Impostare parametri di comunicazione: baud rate, bits di dati, parity, stop bit.
- Inviare stringhe dal microcontrollore al PC (es. messaggio di benvenuto).
- Ricevere comandi dal PC per controllare il LED (es. “ON”/“OFF”).

## Teoria di Base: Comunicazione Seriale Asincrona (UART)

La comunicazione seriale asincrona (UART, Universal Asynchronous Receiver Transmitter) è uno dei metodi più semplici e diffusi per far dialogare microcontrollori e PC.

Ciascun byte di dati viene trasmesso come sequenza di bit sul canale Tx ed encapsulato in un frame composto da:

- **start bit** (livello “0”),
- **bo - b7 data bit**,
- eventuale **bit di parità** (None, Even, Odd),
- 1 o 2 **stop bit** (livello “1”).



La velocità di trasmissione è misurata in bit al secondo e prende il nome di *Baud rate*. (es. 9600, 115200 bps). Deve essere impostato allo stesso valore sia sul trasmettitore che sul ricevitore.

Linee Tx/Rx della porta USART2 (hardware) sono assegnate rispettivamente ai pin PA2 (USART2\_TX) e PA3 (USART2\_RX).

## Modalità di Gestione della Ricezione

Sono possibili due modalità di ricezione dei byte:

- **Polling:** La CPU chiama ripetutamente la funzione `HAL_UART_Receive()` (o controlla il flag `RXNE`) in un loop. Semplice da implementare, ma tiene occupata la CPU anche in assenza di dati.
- **Interrupt:** Si attiva un interrupt alla ricezione di ogni byte (flag `RXNE`), eseguendo una callback (`HAL_UART_RxCpltCallback`). La CPU può svolgere altre attività e viene interrotta solo quando arrivano nuovi dati.

In alternativa, per throughput elevato, ossia per una elevata quantità di dati trasmessi, si può sfruttare la modalità DMA per trasferimenti a blocchi senza intervento continuo della CPU. Questo argomento esula dagli scopi di questo libro.

Nei paragrafi successivi vedremo passo-passo come:

- Abilitare e parametrizzare USART2 in STM32CubeIDE,
- Scrivere il codice di inizializzazione in `MX_USART2_UART_Init()`,
- Inviare un messaggio di benvenuto al boot,
- Gestire la ricezione di comandi via interrupt per accendere/spegnere il LED.

## Configurare il Progetto Serial Polling

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *Serial\_polling*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

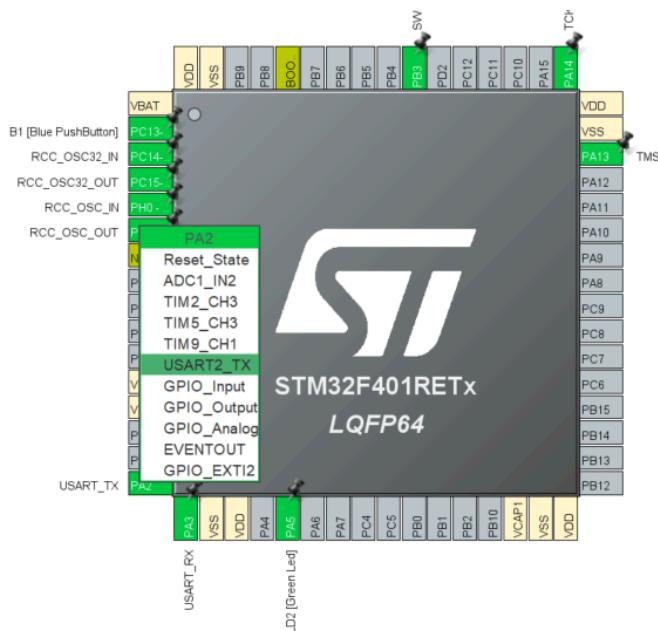
## Configurazione dei pin della USART2

I passi per configurare i pin di USART2 (PA2=Tx, PA3=Rx) in STM32CubeMX sono i seguenti:

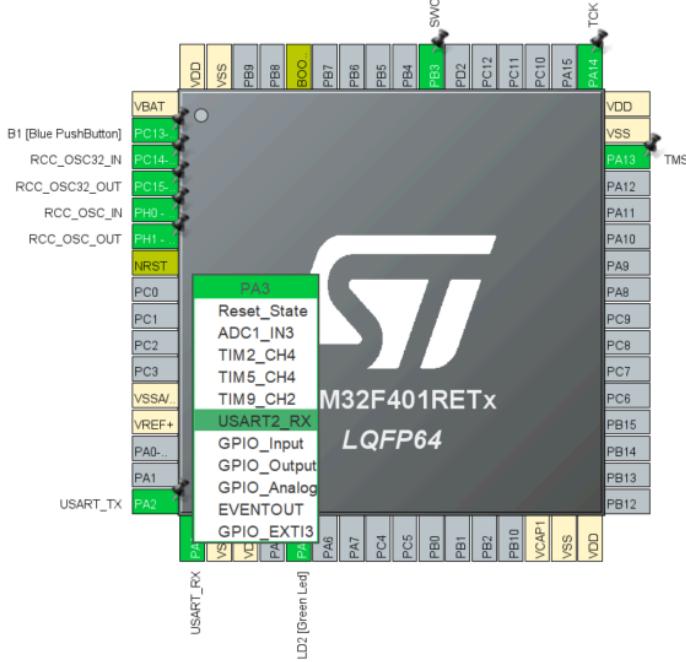
Con un doppio click sul file *Serial\_polling.ioc*, apri il file di configurazione e selezione *Pinout & Configuration Tool*. Individua i pin associati alla porta USART2, PA2 e PA3.

Avendo scelto di eseguire una configurazione di default in fase di creazione del progetto, la porta potrebbe già essere configurata risultando i pin di colore verde.

Per il pin PA2 va selezionata la alternate function *USART2\_TX*



Invece per il pin PA3 va selezionata la alternate function *USART2\_RX*.

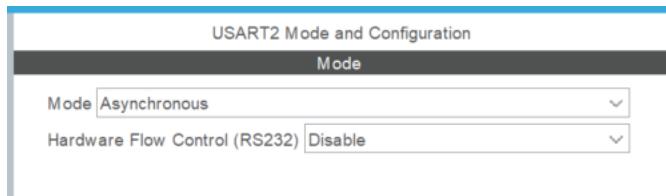


Avendo configurato le alternate function relative ai pin della porta USART2, possiamo passare a configurare i parametri relativi alla comunicazione.

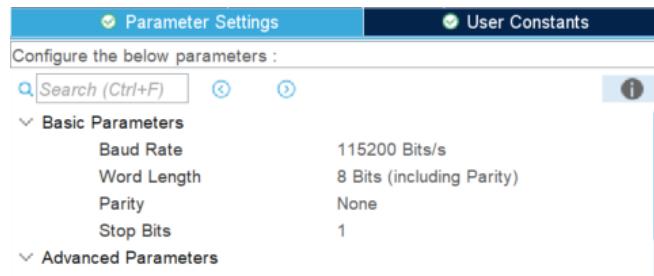
Nel menù *Connectivity*, selezioniamo *USART2*



Scegliamo la modalità asincrona senza abilitare il controllo di flusso hardware.



Infine come parametri selezioniamo:



Configuriamo infine il LED come visto nel capitolo precedente.

Salviamo il progetto ed accettiamo la generazione automatica del codice

A questo punto CubeMX avrà generato la funzione `MX_USART2_UART_Init()` ed instanziato l'handle `huart2` pronto per essere usato.

```
UART_HandleTypeDef huart2;
```

Definiamo il messaggio da inviare al client ed il buffer per la ricezione del comando. Inseriamo il seguente codice nelle sezioni dedicate all'utente.

```
/* 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 */
```

Includiamo la libreria per la gestione delle stringhe. Lo facciamo nella sezione delle private includes

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

Nel ciclo while inseriamo il seguente codice

```

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
}

```

Per inviare il messaggio di prompt utilizziamo la funzione `HAL_UART_Transmit()`

I suoi parametri sono:

```

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`: indica a quale periferica UART applicare la trasmissione. L'handle contiene le impostazioni (baud rate, word length, parity, ecc.) configurate in `MX_USART2_UART_Init()`.
- `(uint8_t*)prompt`: è il puntatore al testo da inviare, che originariamente è una stringa C (`char *`). Il cast a `uint8_t*` serve perché HAL si aspetta un buffer di byte (`uint8_t`).
- `strlen(prompt)`: calcola quanti byte (caratteri) ci sono nel prompt, senza contare il terminatore '`\0`'. Così la funzione trasmette esattamente quel numero di byte.
- `HAL_MAX_DELAY`: valore di timeout “infinito” (di fatto blocca la CPU finché tutti i byte non sono stati inviati). Se la trasmissione non fosse possibile entro questo

tempo, la funzione restituirebbe un errore.

La funzione `memset()` serve a “pulire” (azzerare) un’area di memoria prima di usarla, evitando che rimangano valori “sporchi” (residui) da precedenti operazioni.

La sua firma è:

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

e i parametri sono:

- **rxBuf**: puntatore all’inizio dell’array da azzerare.
- **0**: valore byte da scrivere in ciascuna posizione (qui zero).
- **sizeof(rxBuf)**: numero di byte su cui applicare la scrittura (la dimensione total dell’array).

Quindi:

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

significa “riempi tutti i `sizeof(rxBuf)` byte di `rxBuf` con zeri”. In pratica azzeri l’intero buffer `rxBuf`, impostando ogni elemento a `0`, prima di ricevere il nuovo comando via UART. Questo garantisce che, anche se il comando ricevuto fosse più corto o non terminato, le posizioni rimanenti restino a zero (`'\\0'`), evitando letture di valori casuali.

Questo frammento di codice serve a **filtrare** i caratteri ricevuti e a **prevenire l’overflow** del buffer `rxBuf`. Analizziamo le tre condizioni:

```
if (rxChar != '\r'          // 1) Non memorizzare il carriage return
    && rxChar != '\n'        // 2) Non memorizzare il line feed
    && idx < (sizeof(rxBuf) - 1) // 3) Assicurarsi di lasciare
                                  spazio
                                  per il terminatore '\0'
{
    rxBuf[idx++] = rxChar;   // 4) Memorizza il carattere valido e
                            incrementa l'indice
}
```

- **rxChar != '\r'** Scarta il carattere di “ritorno a capo” (`carriage return`, ASCII 0x0D) inviato da molti terminali quando premi Enter.
- **rxChar != '\n'** Scarta il carattere di “avanzamento linea” (`line feed`, ASCII 0x0A), anch’esso generato al termine della riga.
- **idx < (sizeof(rxBuf) - 1)** Garantisce che l’indice `idx` resti sempre **minore**

della lunghezza del buffer meno uno, in modo tale da riservare un byte per il terminatore nullo ('\0') alla fine della stringa. Questo evita di scrivere oltre i limiti dell'array (**buffer overflow**).

- **rxBuf[idx++] = rxChar;** Se tutte le condizioni sono vere, il carattere ricevuto viene memorizzato in `rxBuf` alla posizione `idx` e poi `idx` viene incrementato (`idx++`), preparandosi a scrivere al prossimo slot disponibile.

In sintesi, questo `if` si assicura di salvare soltanto i caratteri significativi del comando (es. "O", "N", "F", "F") nel buffer, ignorando i terminatori di riga, e di non superare la capacità dell'array riservata per la stringa.

La chiamata della funzione

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

usa l'API HAL in **modalità bloccante** per leggere dati dalla periferica UART. In dettaglio:

I parametri sono

- **&huart2:** Puntatore all'**handle** di USART2, che contiene tutte le configurazioni (baud rate, formattazione frame, ecc.) e lo stato della periferica.
- **rxBuf:** Puntatore al buffer di ricezione, cioè l'array in cui verranno memorizzati i byte ricevuti. Deve essere sufficientemente grande da contenere il numero di byte che ci si aspetta.
- **3:** Numero di byte da ricevere. La funzione resterà in attesa finché non ha ricevuto esattamente 3 caratteri (o fino al timeout).
- **HAL\_MAX\_DELAY:** Timeout massimo in millisecondi per la ricezione: usando `HAL_MAX_DELAY` la chiamata non scade mai, bloccando l'esecuzione finché i 3 byte non sono stati accodati dal driver UART.

### Cosa succede al runtime?

- Il driver UART attiva il ricevimento e aspetta che arrivino 3 byte in ingresso.
- Ogni volta che arriva un bit, l'hardware lo ricostruisce nel byte corrispondente e lo deposita nel registro di ricezione.
- `HAL_UART_Receive` trasferisce ciascun byte dal registro hardware al buffer software (`rxBuf`).
- Solo dopo aver letto tutti e 3 i byte, la funzione restituisce il controllo al programma, che prosegue eseguendo la riga successiva.

Se per qualche motivo i byte non arrivano (ad esempio linea scollegata), il codice rimane

bloccato su questa chiamata fino all'arrivo dei dati. Se volessi invece gestire un timeout, potresti sostituire `HAL_MAX_DELAY` con un valore esplicito di millisecondi.

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

serve a trasformare l'array di byte `rxBuf` in una vera e propria “stringa C” **terminata** dal carattere nullo, in modo che tutte le funzioni di manipolazione stringhe (come `strcmp`, `strlen`, ecc.) sappiano dove fermarsi.

Questo blocco di codice confronta la stringa ricevuta via UART (`rxBuf`) con i comandi testuali “ON” e “OFF” e, in base al risultato, accende o spegne il LED collegato a 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` confronta due stringhe C carattere per carattere.
  - Restituisce **0** se le stringhe sono esattamente uguali.  
Qui si controlla se `rxBuf` (cast a `char*`) è identica alla stringa “ON”.
- `HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);`
  - Se la condizione è vera (ricevuto “ON”), chiama la HAL per portare il pin PA5 a livello alto, accendendo il LED (macro `GPIO_PIN_SET`).
- `else if (strcmp((char*) rxBuf, "OFF") == 0)`
  - Se la prima condizione è falsa, controlla se `rxBuf` corrisponde a “OFF”.
- `HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);`
  - Se ha ricevuto “OFF”, imposta PA5 a livello basso (`GPIO_PIN_RESET`), spegnendo il LED.

Se il contenuto di `rxBuf` non è né “ON” né “OFF”, il codice entra in nessuno dei due blocchi e semplicemente salta al ciclo successivo per ricevere un nuovo comando.

Compilare e scaricare il programma sulla scheda Nucleo.

Per interagire con la Nucleo è necessario aprire un client seriale.

## Configurare il Progetto Serial Interrupt

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *Serial\_interrupt*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

### Configurazione dei pin della USART2

La porta USART2 va configurata come l'esempio precedente. Anche la configurazione del led non differisce dai casi visti in precedenza.

Nel caso di interrupt, il controllo sul comando viene fatto all'interno della funzione di callback che viene chiamata ogni volta che un byte è stato ricevuto.

La funzione di callback che il programmatore deve implementare è

```
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);
    }
}
```

La ricezione tramite interrupt viene avviata invece attraverso l'invocazione della funzione HAL

```
HAL_UART_Receive_IT(&huart2, &rxChar, 1);
```

# Capitolo 5 GPIO input - polling

In questo progetto, impareremo a configurare un pin GPIO come input digitale per leggere lo stato del pulsante utente della scheda Nucleo. Utilizzeremo il pulsante per accendere e spegnere il led verde.

## Obiettivi del Progetto

- Comprendere il funzionamento di un pin GPIO come input digitale.
- Configurare un progetto base in STM32CubeIDE.
- Leggere lo stato del pulsante blu della NucleoF401RE utilizzando codice C.

## Teoria di Base: Lettura di un GPIO di input

Quando si intende utilizzare un pin GPIO come input digitale, va tenuto conto delle due modalità di lettura:

- **Polling:** Il microcontrollore controlla continuamente lo stato del GPIO in un ciclo ripetitivo (loop). È un metodo sincrono, la CPU è costantemente impegnata a leggere il pin, anche quando lo stato non cambia.
- **Interrupt:** Il microcontrollore esegue il codice principale e "si ferma" solo quando il GPIO cambia stato, generando un segnale di interrupt. È un metodo asincrono, la CPU si occupa di altre attività fino a quando non viene notificata dal GPIO.

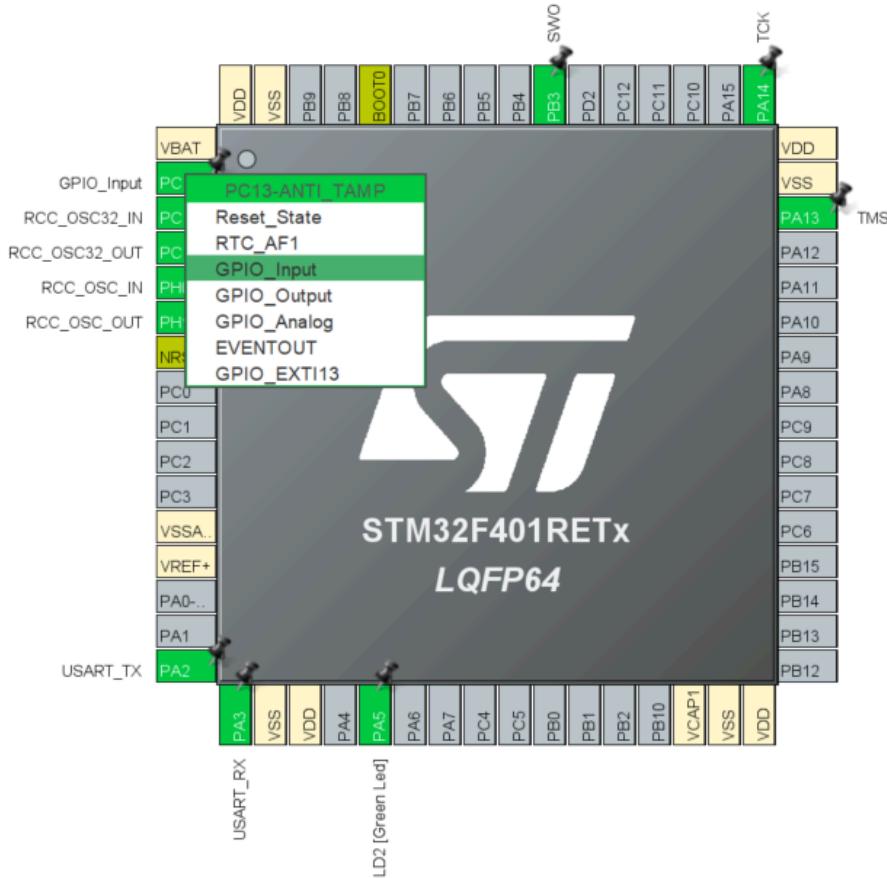
## Configurare il Progetto

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *UserButton\_polling*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

## Configurazione del pin User Button

Apri il Pinout & Configuration Tool ed individua il pin associato al pulsante utente sulla scheda NucleoF401RE, lo User Button è collegato al pin PC13.

Clicca sul pin PC13 sulla mappa del microcontrollore e impostalo come GPIO\_Input. Il pin potrebbe già essere stato configurato da CubeMX.



Volendo fare accendere il led, è necessario configurare anche il pin corrispondente come visto in precedenza, assegnando al pin del led l'etichetta *GREEN\_LED*.

Nella scheda **GPIO Configuration**, selezionare il pin **PC13** e configurare i seguenti parametri:

- **Mode:** Input Mode
- **Pull-up/Pull-down:** Pull-up (opzionale, ma spesso raccomandato per migliorare la stabilità del pulsante, in quanto il pulsante collega il pin a GND quando premuto).
- **Label:** Puoi optionalmente assegnare un'etichetta come *USER\_BTN*.

The screenshot shows the STM32CubeIDE interface for configuring GPIO pins. On the left, there's a sidebar with categories like System Core, Analog, Timers, Connectivity, Multimedia, Computing, and Middleware and Software Packs. The 'System Core' section has 'GPIO' selected. The main area is titled 'GPIO Mode and Configuration' and shows a table for pin configuration. The table includes columns for Pin Name, Signal on Pin, GPIO output, GPIO mode, GPIO Pull..., Maximum..., User Label, and Modified. Two rows are visible: PA5 (n/a, Low, Output Push..., No pull-up a..., Low, GREEN\_LED) and PC13-ANTI\_TAMP (n/a, n/a, External Int..., Pull-up, n/a, USER\_BTN). Below the table, there's a detailed configuration panel for the PC13-ANTI\_TAMP pin, showing options for GPIO mode (External Interrupt Mode with Falling edge trigger detection), GPIO Pull-up/Pull-down (Pull-up), and User Label (USER\_BTN).

Completata la configurazione dei pin, possiamo generare il codice. Per farlo sarà sufficiente salvare il progetto, oppure in generale cliccare su **Project -> Generate Code**.

STM32CubeIDE genererà il codice necessario per configurare il pin in modalità input.

## Lettura del Pulsante

Nel file main.c (o main.cpp per un progetto C++), nel loop principale (while(1)), leggiamo lo stato del pulsante utilizzando la funzione HAL\_GPIO\_ReadPin.

Quando viene premuto il pulsante il pin è portato allo stato basso codificato dalla macro GPIO\_PIN\_RESET. Quindi dovremo introdurre un controllo per verificare che il pin abbia assunto questo valore.

Per accendere il led, utilizziamo la funzione di toggle dei pin associazion al led verde.

Di seguito il codice da inserire nel loop nella sezione USER CODE.

```
/* 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 */
}
```

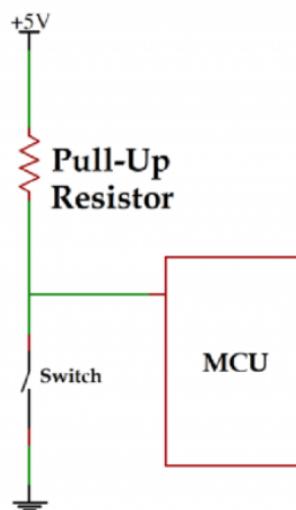
Questo codice ha il solo scopo educativo, nella pratica andrebbe migliorato aggiungendo il *debouncing*.

Compilare e scaricare il programma sulla scheda Nucleo.

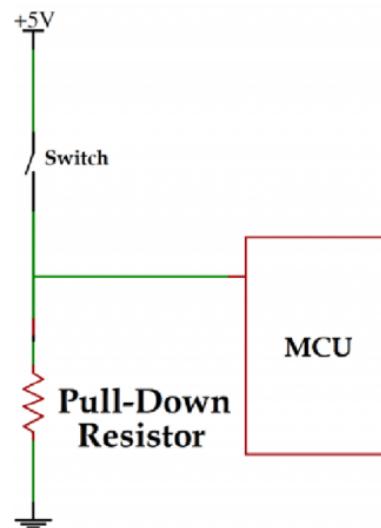
## Approfondimento: I resistori di pull up/down

I resistori pull-up (o pull-down) servono a garantire la stabilità del segnale di un pin in uno stato ben definito (alto o basso) quando non è attivamente collegato a una sorgente esterna, come nel caso di un pulsante non premuto.

- **Comportamento senza resistore pull-up:** Quando il pulsante non è premuto, il pin è scollegato (in un circuito "aperto") e il suo stato logico è indeterminato. Questo fenomeno è noto come **stato flottante** (floating). Un pin in stato flottante può rilevare segnali di rumore o variazioni casuali nella tensione, portando a letture instabili o imprevedibili.
- **Funzione del resistore pull-up:** Il resistore pull-up è collegato tra il pin di input e la tensione positiva di alimentazione (+Vcc). Quando il pulsante è rilasciato (non premuto), il resistore garantisce che il pin venga "tirato su" verso lo stato logico alto (HIGH). Quando il pulsante viene premuto, collega direttamente il pin a GND, forzandolo allo stato logico basso (LOW).



- **Stabilità del pulsante:** Con un resistore pull-up, il circuito garantisce un passaggio chiaro e stabile tra lo stato HIGH (rilasciato) e LOW (premuto). Questo elimina il rumore e gli stati indeterminati. La stessa logica si applica al resistore pull-down, che invece "tira" il pin verso GND, garantendo lo stato LOW quando il pulsante non è premuto.



Molti microcontrollori moderni come STM32, offrono resistori pull-up/pull-down interni configurabili via software

# Capitolo 6 GPIO input - interrupt

In questo progetto, impareremo a configurare un pin GPIO per generare un interrupt digitale in risposta alla pressione del pulsante utente della scheda Nucleo. Utilizzeremo questo interrupt per accendere e spegnere il LED verde in modo reattivo ed efficiente.

## Obiettivi del Progetto

- Comprendere il funzionamento degli interrupt associati a un pin GPIO.
- Configurare un progetto base in CubeIDE con gestione degli interrupt.
- Utilizzare un interrupt per rilevare la pressione del pulsante blu della NucleoF401RE e gestire eventi tramite codice C.

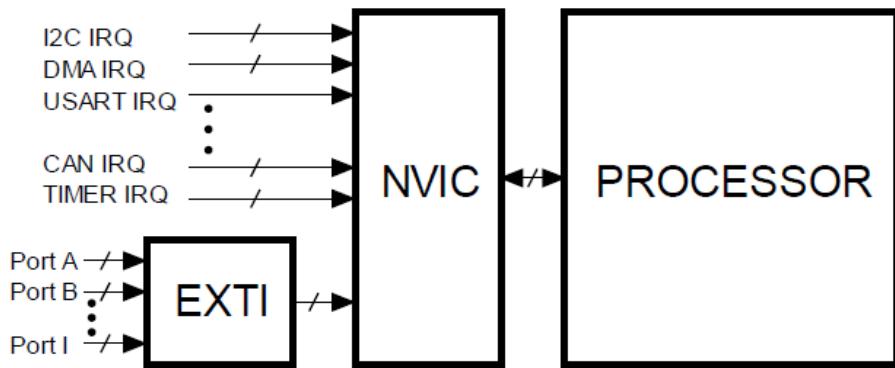
## Teoria di Base: Generazione di un interrupt

Un interrupt è un segnale che permette al microcontrollore di interrompere l'esecuzione del codice principale per rispondere a un evento specifico, come il cambio di stato di un pin GPIO. Questo approccio è asincrono, la CPU può continuare a eseguire altre attività fino a quando non viene notificata dell'evento tramite l'interrupt.

Nel microcontrollore STM32F401RE, la gestione degli interrupt è affidata al **NVIC (Nested Vectored Interrupt Controller)**, che prioritizza e smista gli interrupt. Quando un pin GPIO configurato per generare un interrupt rileva un cambiamento di stato (ad esempio, un fronte di salita o di discesa), il microcontrollore:

- **Rileva l'evento:** Il sistema GPIO registra il cambiamento di stato configurato come trigger (es. fronte di salita o discesa).
- **Invia la richiesta al NVIC:** Il GPIO notifica al NVIC che un evento di interrupt è avvenuto.
- **Esegue la routine ISR (Interrupt Service Routine):** La CPU sospende temporaneamente il codice principale per eseguire la funzione associata all'interrupt (ISR), che gestisce l'evento.
- **Ritorna al codice principale:** Dopo aver completato l'ISR, la CPU riprende da dove era stata interrotta.

Questo sistema garantisce una risposta veloce agli eventi senza compromettere l'efficienza complessiva, poiché la CPU non deve continuamente controllare lo stato del GPIO (metodo polling).



Lo schema in figura rappresenta il flusso e la gestione degli interrupt nei microcontrollori STM32, utilizzando i blocchi principali coinvolti: EXTI, NVIC, e il PROCESSOR.

- **EXTI** (External Interrupt/Event Controller): Questo blocco è responsabile della gestione degli interrupt generati dai pin GPIO. I segnali di interrupt possono provenire da varie *port* (es. Port A, Port B, fino a Port I), ovvero dai pin configurati per generare eventi di interrupt. Il compito dell'EXTI è identificare il pin che ha generato l'interrupt e notificare l'NVIC.
- **NVIC** (Nested Vectored Interrupt Controller): Questo è il componente principale che gestisce e priorizza tutti gli interrupt del sistema. Riceve richieste sia dall'EXTI (interrupt esterni) sia da altre periferiche del microcontrollore, come:
  - **I2C IRQ**: Interrupt generato dal modulo I2C.
  - **DMA IRQ**: Interrupt legato ai trasferimenti diretti di memoria (DMA).
  - **USART IRQ**: Interrupt del modulo di comunicazione seriale USART.
  - **CAN IRQ**: Interrupt del modulo di comunicazione CAN.
  - **TIMER IRQ**: Interrupt generato dai timer.
- Il NVIC assegna una priorità agli interrupt, per garantire che quelli più critici vengano gestiti prima. Inoltre, è in grado di gestire interrupt annidati, ossia sospendere un ISR (Interrupt Service Routine) per eseguire un altro interrupt con priorità più alta.
- **PROCESSOR**: Questo è il core del microcontrollore, responsabile dell'esecuzione del codice principale. Quando l'NVIC notifica un interrupt, il PROCESSOR sospende temporaneamente il flusso normale del programma e passa a eseguire l'Interrupt Service Routine (ISR), ovvero la funzione di gestione dell'interrupt. Una volta terminata l'ISR, il PROCESSOR riprende il codice principale da dove era stato interrotto.

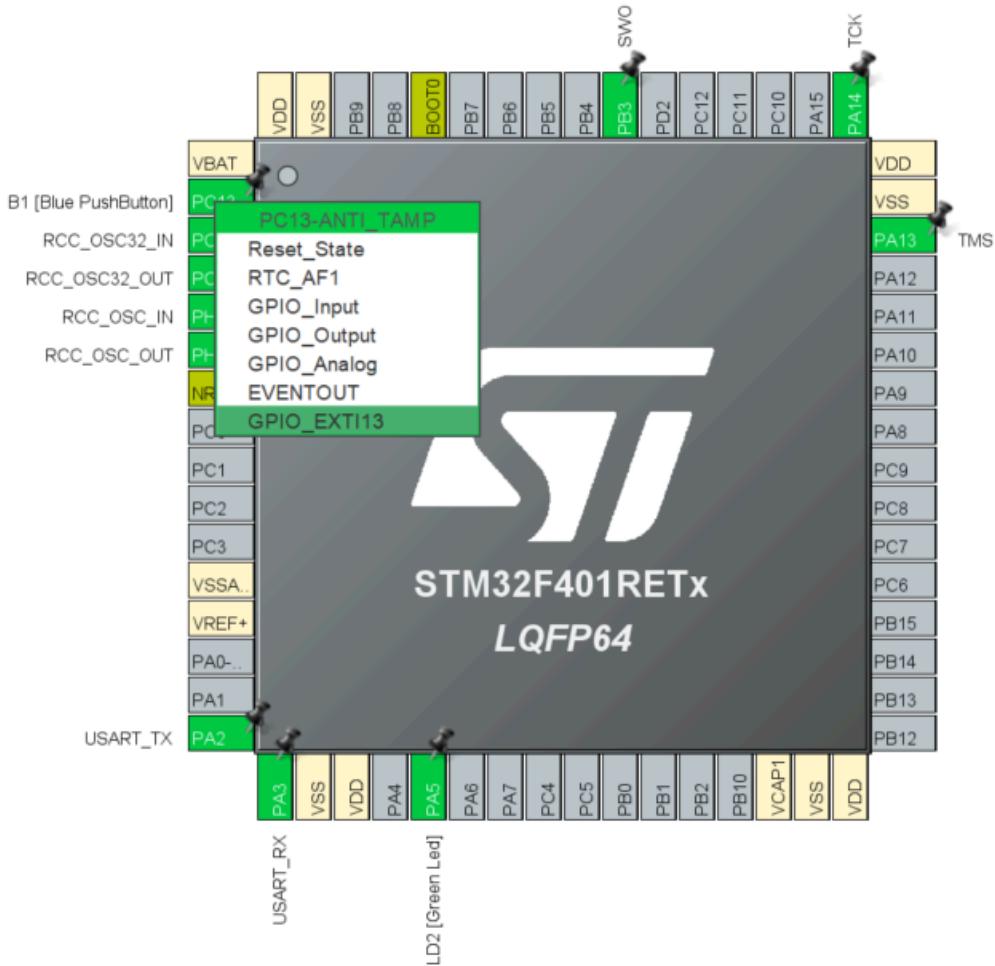
# Configurare il Progetto

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *UserButton\_interrupt*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

## Configurazione del pin User Button

Apri il Pinout & Configuration Tool ed individua il pin associato al pulsante utente sulla scheda NucleoF401RE, lo User Button è collegato al pin PC13.

Clicca sul pin PC13 sulla mappa del microcontrollore e impostalo come GPIO\_EXTI13. Il pin potrebbe già essere stato configurato da Cube.



Volendo fare accendere il led, è necessario configurare anche il pin corrispondente come visto in precedenza, assegnando al pin del led l'etichetta *GREEN\_LED*.

Nella scheda **GPIO Configuration**, selezionare il pin **PC13** e configurare i seguenti parametri:

- **Mode:** External Interrupt Mode with Falling Edge trigger detection
- **Pull-up/Pull-down:** Pull-up (opzionale, ma spesso raccomandato per migliorare la stabilità del pulsante, in quanto il pulsante collega il pin a GND quando premuto).
- **Label:** Puoi opzionalmente assegnare un'etichetta come USER\_BTN.

PC13-ANTI\_TAMP Configuration :

GPIO mode	External Interrupt Mode with Falling edge trigger detection
GPIO Pull-up/Pull-down	Pull-up
User Label	USER_BTN

A questo punto non resta che abilitare l'interrupt.

- Nella scheda **NVIC Settings** (nella configurazione di GPIO o nella scheda principale di configurazione).
- Abilita l'interrupt per **EXTI Line[15:10]** e imposta la priorità desiderata.

The screenshot shows the NVIC Interrupt Table configuration in STM32CubeMX. The table has columns for NVIC Interrupt Table, Enabled, Preemption Priority, and Sub Priority. The 'EXTI line[15:10] interrupts' row is highlighted, indicating it is being configured. The 'Enabled' column for this row contains a checked checkbox, while other rows have unchecked checkboxes.

Interrupt Source	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	<input checked="" type="checkbox"/>	0	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0	0
Memory management fault	<input checked="" type="checkbox"/>	0	0
Pre-fetch fault, memory access fault	<input checked="" type="checkbox"/>	0	0
Undefined instruction or illegal state	<input checked="" type="checkbox"/>	0	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0	0
Debug monitor	<input checked="" type="checkbox"/>	0	0
Pendable request for system service	<input checked="" type="checkbox"/>	0	0
Time base: System tick timer	<input checked="" type="checkbox"/>	0	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0	0
Flash global interrupt	<input type="checkbox"/>	0	0
RCC global interrupt	<input type="checkbox"/>	0	0
USART2 global interrupt	<input type="checkbox"/>	0	0
EXTI line[15:10] interrupts	<input checked="" type="checkbox"/>	2	0
FPU global interrupt	<input type="checkbox"/>	0	0

La **priority** e la **sub-priority** degli interrupt sono due livelli di configurazione offerti dal controller NVIC (Nested Vectored Interrupt Controller) nei microcontrollori STM32. Questi permettono di definire quali interrupt devono essere gestiti prima in caso di conflitti o eventi simultanei. Per questo progetto possiamo considerare priorità 2 per l'interrupt generato dal pulsante.

Salviamo il progetto e generiamo il codice.

## Gestione del pulsante

La funzione di inizializzazione contiene la definizione del pin *USER\_BTN* e l'abilitazione dell'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);
```

Rimane da implementare la funzione di callback, che viene invocata automaticamente quando si verifica un interrupt.

La funzione da definire nel file *main.c* è *HAL\_GPIO\_EXTI\_Callback(uint16\_t GPIO\_Pin)*. Questa funzione è progettata per gestire l'evento associato all'interrupt generato. Nel nostro progetto, l'obiettivo è modificare lo stato del LED verde ogni volta che viene premuto il pulsante. Per farlo, utilizzeremo la funzione di toggle sul pin GPIO associato al LED verde, invertendone lo stato ad ogni chiamata della callback.

```

/* 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 */

```

Compilare e scaricare il programma sulla scheda Nucleo

## Approfondimento: Priorità degli Interrupt - Concetti di Base

Quando si verifica un interrupt, il microcontrollore deve decidere quale ISR (Interrupt Service Routine) eseguire. Questo viene determinato in base alla priorità e alla sub-priority configurate nel NVIC.

La priorità principale stabilisce quale interrupt ha la precedenza rispetto agli altri. Ad esempio:

- Un interrupt con priorità 1 ha la precedenza su uno con priorità 2.
- Se due interrupt con priorità diversa si verificano contemporaneamente, il microcontrollore esegue prima quello con priorità più alta (numero più basso).

La sub-priority viene utilizzata per gestire l'ordine di esecuzione di interrupt che hanno la stessa priorità principale. In caso di interrupt simultanei con uguale priorità principale, il microcontrollore usa la sub-priority per decidere quale ISR eseguire prima.

In STM32, il sistema NVIC supporta fino a 16 livelli di priorità principale (a seconda del dispositivo) e più livelli di sub-priority.

Il numero di livelli di priorità principale e sub-priority è determinato dal Priority Group configurato. Questo configura il numero di bit assegnati per la priority e la sub-priority.

Il Priority Grouping divide i bit disponibili per la priorità in due campi:

- **Bits per Priorità Principale:** Determinano il livello di priorità principale.
- **Bits per Sub-Priority:** Determinano la priorità relativa tra interrupt dello stesso gruppo principale.

L'assegnazione viene configurata con la funzione:

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

I gruppi disponibili sono:

<b>Priority Group</b>	<b>Bits per Priorità Principale</b>	<b>Bits per Sub-Priority</b>
NVIC_PRIORITYGROUP_0	0	Tutti i bit per sub-priority
NVIC_PRIORITYGROUP_1	1	3
NVIC_PRIORITYGROUP_2	2	2
NVIC_PRIORITYGROUP_3	3	1
NVIC_PRIORITYGROUP_4	4	0 (solo priorità principale)

Se non viene definito il priority group, il valore di default tipico è il GROUP\_2 che assegna due bit per il priority e due per il sub-priority. Ciò evidentemente significa che saranno possibili  $2^2 = 4$  priority e  $2^2 = 4$  sub-priority.

# Capitolo 7 Analog input - ADC

In questo progetto, impareremo a configurare un pin GPIO per leggere un segnale analogico. Utilizzeremo un potenziometro per accendere e spegnere il LED verde a seconda se il valore letto superi o meno una soglia predefinita.

## Obiettivi del Progetto

- Comprendere il funzionamento del convertitore Analogico-Digitale (ADC) su un microcontrollore.
- Configurare un progetto base in CubeIDE per la lettura di segnali analogici.
- Utilizzare un potenziometro per controllare il valore di un segnale analogico e implementare una logica decisionale per accendere o spegnere un LED in base a una soglia predefinita.
- Configurazione della porta UART per l'invio di messaggi

## Teoria di Base: Lettura di un Segnale Analogico

Un segnale analogico è un segnale continuo che può assumere un numero infinito di valori all'interno di un intervallo. Per elaborarlo, il microcontrollore utilizza un convertitore Analogico-Digitale (ADC), che trasforma il segnale analogico in un valore digitale interpretabile dal codice. Configurando opportunamente il modulo ADC, è possibile leggere il valore generato da un potenziometro collegato a un pin analogico e utilizzare questo valore per gestire comportamenti specifici, come l'accensione o lo spegnimento di un LED.

## Cos'è un ADC (Analog-to-Digital Converter)?

Un ADC (Convertitore Analogico-Digitale) è un componente elettronico che trasforma segnali analogici (valori continui) in segnali digitali (valori discreti) interpretabili da un microcontrollore. Nel caso del STM32F401RE, il microcontrollore è dotato di un ADC a 12 bit, che consente di rappresentare il valore analogico in un intervallo da 0 a 4095.

Le principali caratteristiche dell'ADC nel STM32F401RE sono:

- **Risoluzione:** 12 bit (può essere configurata anche a 6, 8 o 10 bit per ridurre i tempi di conversione).
- **Canali:** Fino a 16 canali, che permettono di acquisire segnali da più ingressi analogici.
- **Modalità di conversione:**
  - Singola conversione (Single Conversion Mode).
  - Conversione continua (Continuous Conversion Mode).
  - Conversione regolare e/o in modalità iniettata.
- **Timer trigger:** Possibilità di sincronizzare le conversioni con un timer interno.
- **DMA support:** Il trasferimento dei dati può essere automatizzato tramite DMA (Direct Memory Access).

## Cos'è un canale ADC?

Un canale ADC è un ingresso fisico o logico associato a un pin del microcontrollore. Ogni canale corrisponde a una specifica sorgente analogica che l'ADC può convertire. I canali disponibili sullo STM32F401RE includono:

- **Ingressi analogici:** Associati a pin GPIO configurati in modalità analogica (ad esempio, PA0 è associato al canale ADC1\_IN0).
- **Canali interni:** Come il sensore di temperatura interno, VREFINT (riferimento interno di tensione) e VBAT (tensione di alimentazione della batteria).

Di seguito è riportata la mappa dei canali del ADC1 del microcontrollore F401RE

Canale ADC	Pin GPIO associato
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

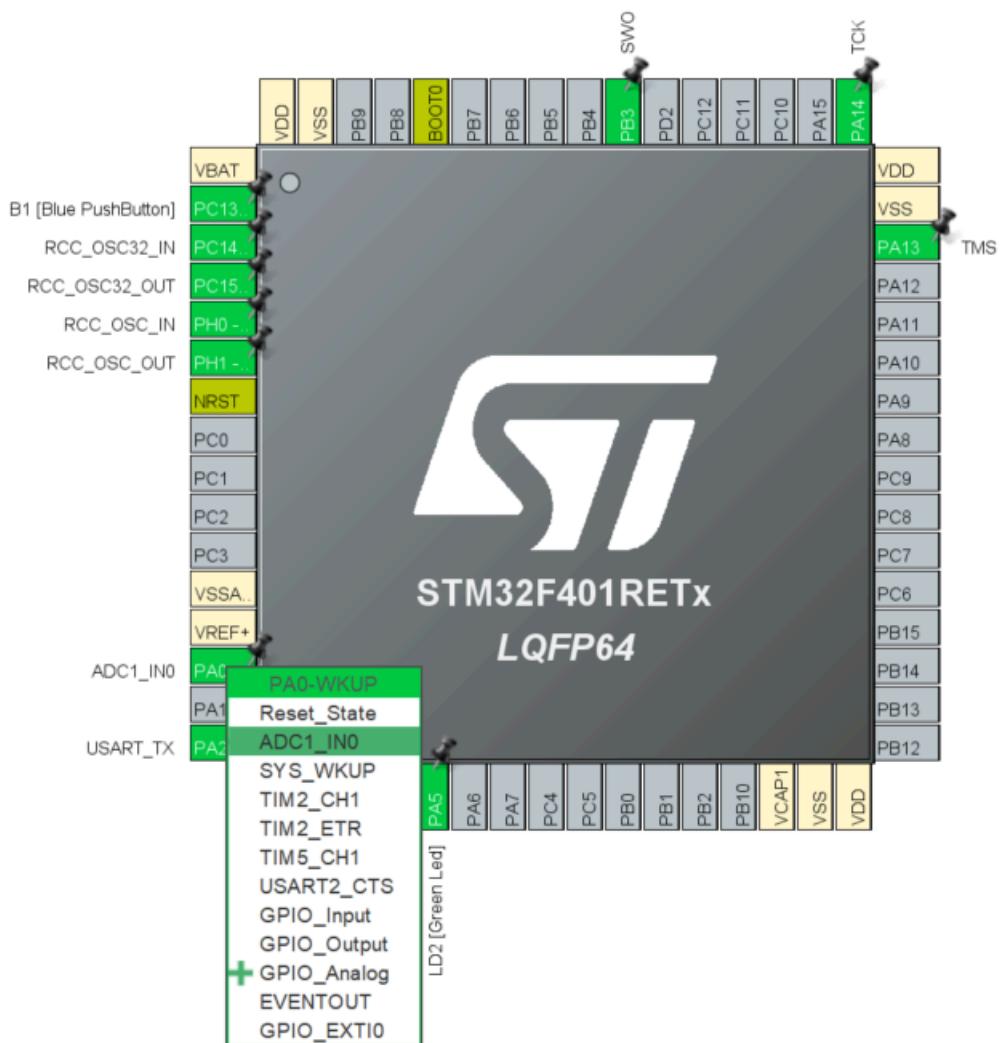
## Configurare il Progetto

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *Analog\_input*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

### Configurazione del pin A0

Apri il Pinout & Configuration Tool ed individua il pin PA0.

Clicca sul pin PA0 sulla mappa del microcontrollore e impostalo come ADC1\_IN0.



Seleziona ADC1, l'unico disponibile per il pin *PA0*.

Categories | A-Z

- System Core >
- Analog <
- ADC1**
- Timers >
- Connectivity >
- Multimedia >
- Computing >
- Middleware and Software Packs >

Ed in *Mode* assicurati che *IN0* sia selezionato. Dobbiamo selezionare *IN0* e non gli altri canali, perché il canale 0 è quello associato al pin *PA0*



In *Parameters Settings* lasciamo tutto invariato tranne il parametro *Sampling Time* poichè volendo utilizzare tutti i 12 bit del convertitore, saranno necessari 15 cicli

Parameter Settings		User Constants
Configure the below parameters :		
<input type="text"/> Search (Ctrl+F)		
Resolution	12 bits (15 ADC Clock cycles)	
Data Alignment	Right alignment	
Scan Conversion Mode	Disabled	
Continuous Conversion Mode	Disabled	
Discontinuous Conversion Mode	Disabled	
DMA Continuous Requests	3 Cycles	
End Of Conversion Selection	15 Cycles	
ADC_Regular_ConversionMode	28 Cycles	
Number Of Conversion	56 Cycles	
External Trigger Conversion Source	84 Cycles	
External Trigger Conversion Edge	112 Cycles	
Rank	144 Cycles	
Channel	480 Cycles	
Sampling Time	3 Cycles	
> ADC_Injected_ConversionMode		
> WatchDog		

La *UART2* ed il *LED* di colore verde sono già inizializzati, avendo chiesto l'inizializzazione ai valori di *default* in fase di creazione del progetto. Qualora fosse necessario, fai riferimento ai capitoli dedicati alla loro inizializzazione.

Per verificare che il led e la *UART2* pin siano inizializzati, sarà sufficiente controllare che i rispettivi pin siano colorati di verde.

Siamo pronti a generare il codice. Lo facciamo salvando il progetto, nel qual caso CubeIDE ci chiederà se lo vogliamo generare, oppure chiedendo esplicitamente nel menù *Project/Generate Code*.

Volendo stampare a video il valore letto dall'ADC, faremo uso di funzioni contenute nelle librerie *stdio* e *string*. Dobbiamo dunque procedere all'inclusione dei rispettivi header file nella sezione degli include previste per lo *USER CODE*.

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

Per utilizzare il valore letto dall'ADC, abbiamo bisogno di due variabili private. Una di tipo *uint16\_t* per memorizzare il valore letto, codificato in 12 bits, ed un array di caratteri per il messaggio da inviare. Li andiamo ad inserire nella sezione allocata alle *Private*

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

Per leggere l'ADC dobbiamo aggiungere il seguente codice nella sezione del main loop.

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

Avvia il convertitore analogico-digitale (ADC) con `HAL_ADC_Start`, in modo da iniziare la conversione del segnale analogico in un valore digitale.

Attende il completamento della conversione usando `HAL_ADC_PollForConversion`, con un timeout di 20 millisecondi per evitare blocchi infiniti.

Recupera il valore convertito tramite `HAL_ADC_GetValue` e lo memorizza nella variabile "measure".

Formatta il valore letto in una stringa utilizzando `sprintf`, preparandola per la trasmissione.

Trasmette il messaggio formattato sulla porta UART (`HAL_UART_Transmit`) con un timeout massimo, assicurando che il messaggio venga inviato correttamente.

Introduce una breve pausa di 50 millisecondi (`HAL_Delay`) prima di eseguire eventuali ulteriori operazioni.

Compilare e scaricare il programma sulla scheda Nucleo.

Per visualizzare i messaggi inviati dalla scheda Nucleo sarà necessario utilizzare un client, aprire la connessione sulla porta assegnata alla Nucleo dal sistema operativo, selezionando 115200 bps come velocità di connessione.

# Capitolo 8 Timers

In questo capitolo approfondiremo l'uso dei timer sul microcontrollore STM32, illustrando i concetti fondamentali e guidandoti nella configurazione pratica del progetto.

## Obiettivi del Progetto

L'obiettivo principale di questo progetto è farti sperimentare e comprendere il funzionamento dei timer in ambiente STM32. In particolare, imparerai a:

- **Generare eventi periodici:** Configurare il timer per eseguire azioni a intervalli regolari.
- **Gestire gli interrupt:** Configurare gli interrupt derivanti dal timer per ottimizzare la gestione delle operazioni senza sovraccaricare il processore.
- **Sperimentare in modo pratico:** Mettere in pratica le conoscenze teoriche con esercitazioni reali che includono la configurazione del timer attraverso strumenti come STM32CubeIDE e l'uso della HAL.

## Teoria di base: Cos'è un Timer

I timer sono periferiche hardware fondamentali per il controllo temporale all'interno dei sistemi embedded, poiché consentono di contare gli impulsi del clock e di misurare intervalli di tempo in modo preciso.

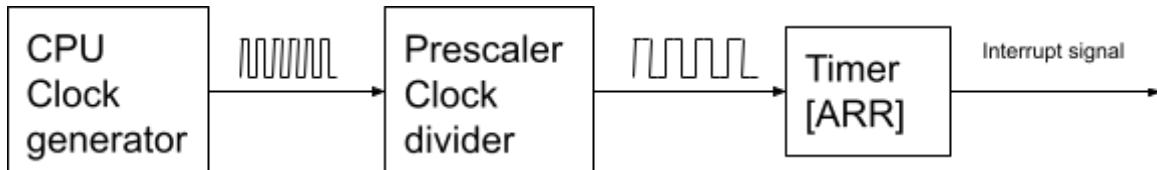
Con un timer possiamo sia misurare intervalli di tempo che generare eventi temporizzati, sfruttando il conteggio degli impulsi di un segnale di clock a frequenza nota. Conoscendo la frequenza del clock e contando il numero di impulsi, è possibile calcolare con precisione l'intervalllo di tempo trascorso.

Utilizzare correttamente un timer implica saper bilanciare la frequenza di clock, il prescaler e il valore di autoreload per ottenere la risoluzione temporale desiderata e garantire una gestione efficiente degli eventi, elemento essenziale per il corretto funzionamento delle applicazioni embedded.

Per far scattare un evento dopo un determinato periodo, si imposta un valore di confronto, che corrisponde al numero di impulsi da contare, e si può regolare l'intervalllo di conteggio mediante l'uso di un prescaler.

Il prescaler riduce la frequenza in ingresso dividendo il clock per un fattore predefinito, permettendo così di ottenere intervalli di tempo più lunghi. Inoltre, il timer dispone di un

registro chiamato ARR (Auto-Reload Register) nel quale viene definito il valore massimo del conteggio: una volta raggiunto questo valore, il contatore si resetta e, se opportunamente configurato, genera un evento (ad esempio, l'attivazione di un interrupt). In questo modo, la combinazione di prescaler e ARR consente di ottenere la risoluzione temporale desiderata e di gestire in modo efficiente le operazioni basate sul tempo.



Inoltre, i timer operano in modalità multiple, tra cui la generazione di segnali PWM, l'input capture per misurare la durata di eventi esterni e l'output compare per gestire eventi a scadenza, offrendo così una flessibilità che permette al microcontrollore di eseguire operazioni temporali in maniera indipendente e reattiva. Questo argomento sarà trattato nel capitolo successivo.

## Configurare il Progetto

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *Timer*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

Con un doppio click sul file *Timer.ioc* accediamo alla configurazione del microcontrollore.

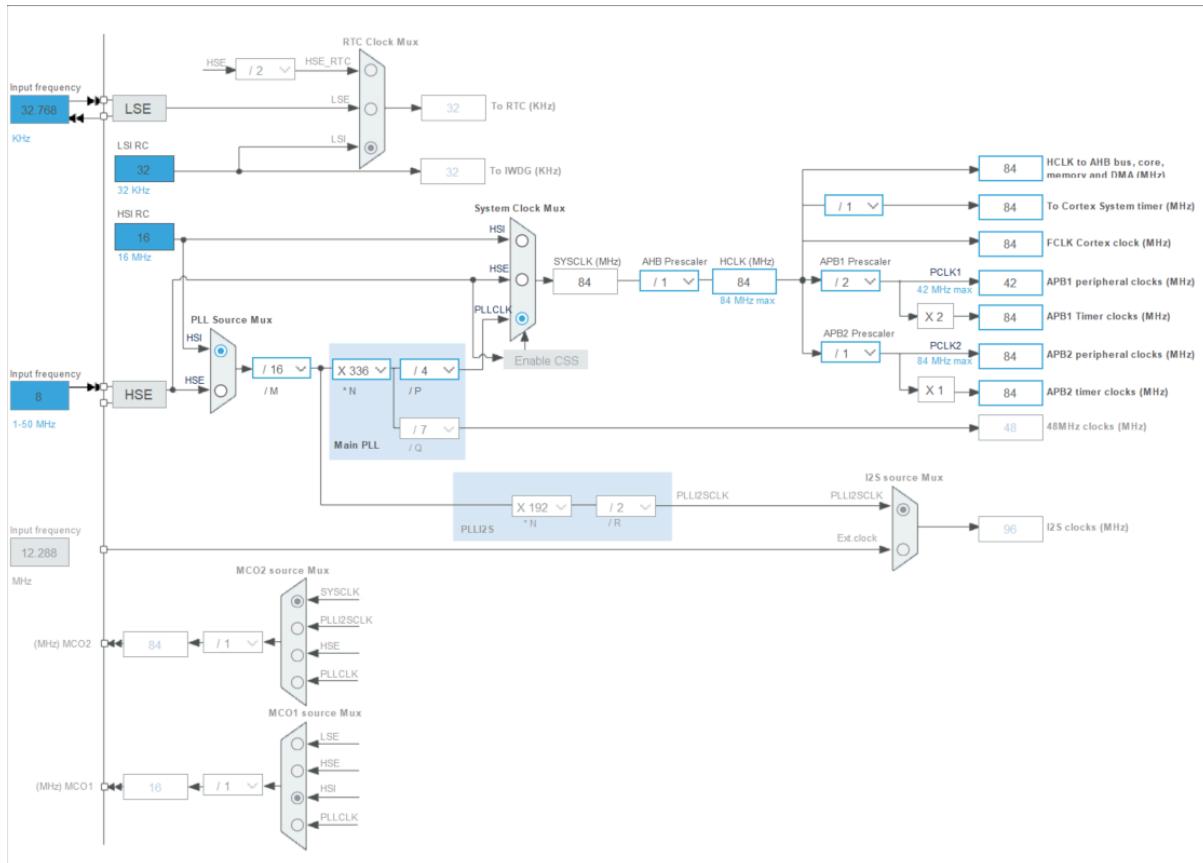
Dalla pagina *Pinout & Configuration* spostiamoci sulla pagina *Clock Configuration*. In questa pagina è possibile effettuare alcune modifiche di configurazione del clock delle periferiche.

Possiamo modificare alcuni di blocchi dell'*albero*, ossia quelli relativi ai prescaler, AHB e APB1 e APB2.

L'AHB prescaler è un divisore di frequenza applicato al clock principale del microcontrollore per ottenere la frequenza operativa dell'Advanced High-performance Bus (AHB). Questo bus collega il core del processore alle periferiche ad alta velocità e, attraverso il prescaler, il clock di sistema viene ridotto in modo da soddisfare le specifiche operative delle diverse parti del dispositivo, migliorando l'efficienza energetica e garantendo la stabilità del sistema.

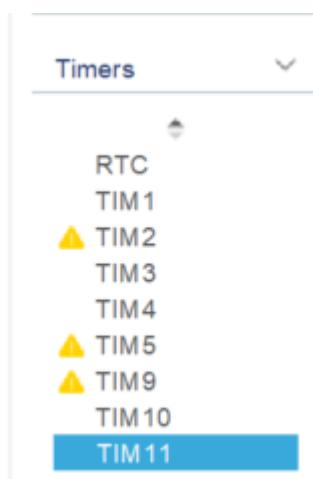
Gli APB (Advanced Peripheral Bus) sono bus interni che collegano il core del microcontrollore alle periferiche, svolgendo un ruolo fondamentale nell'architettura dei sistemi embedded basati su ARM, come gli STM32. In particolare, questi bus sono progettati per gestire la comunicazione con dispositivi che non richiedono elevate velocità di trasferimento dati, come UART, I2C, SPI, timer e ADC. Nei microcontrollori STM32, gli APB sono generalmente suddivisi in due gruppi: APB1, destinato a periferiche a frequenza più bassa, e APB2, che supporta dispositivi che operano a frequenze più elevate. La frequenza di ciascun bus viene ottenuta applicando un prescaler al clock principale (derivato dall'AHB), in modo da adattare la velocità di funzionamento alle specifiche esigenze delle periferiche.

collegate, garantendo così una comunicazione efficiente e ottimizzando il consumo energetico.

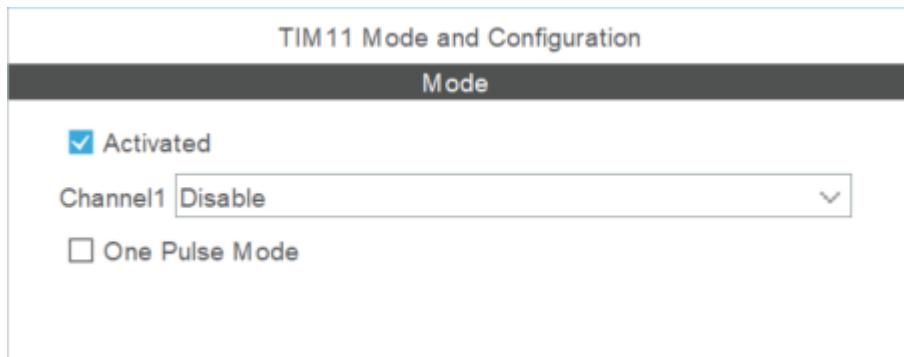


Per il momento non modifichiamo nulla, e torniamo alla pagina di *Pinout & Configuration*.

Selezioniamo *Timers*



e poi *TIM11* ed entriamo nella modalità di configurazione del timer



Spuntiamo il campo *Activated* per abilitare il timer e lasciamo al loro valore di default gli altri campi.

Nei Parameter Settings definiamo il valore del prescaler. Poiché il valore parte da 0 e corrisponde ad un prescaler 1, può essere comodo inserire un'espressione in cui al valore desiderato si sottrae 1. Quindi desiderando un prescaler 84, e dovendo inserire 83, andiamo ad inserire l'espressione 84-1.

Facciamo la stessa cosa per il valore dell'ARR. volendo fare contare il valore massimo, ossia 65536, ma dovendo inserire 65535, scriveremo l'espressione 65536-1.

Parameter	Value
Prescaler (PSC - 16 bits value)	84-1
Counter Mode	Up
Counter Period (AutoReload Register)	65536-1
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

Essendo la frequenza del clock pari ad 84 MHz, ed avendo selezionato un valore di prescaler pari ad 83 (84-1), avremo che il timer vedrà un segnale di frequenza pari a:

$$freq_{desiderata} = 84\text{MHz} \div 84 = 1\text{MHz}$$

$$\text{ossia } 1 \text{ tick ogni } 1 \div 1\text{MHz} = 1\mu\text{s}$$

Avendo inserito ARR = 65536-1 riusciremo al massimo a contare 65536 us ossia 65,536 ms

In generale valgono le seguenti relazioni:

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

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

Per quanto riguarda il *Counter Mode*, il valore *Up* imposta il timer per un conteggio crescente. Infine disabilitiamo l'*auto-reload preload*.

Salviamo il progetto ed accettiamo la generazione del codice.

CubeIDE provvederà alla creazione di una funzione di inizializzazione con i parametri inseriti nel 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 */
```

Volendo inviare tramite porta seriale il tempo trascorso, includiamo la standard IO (*stdio.h*) library

Lo facciamo nella sezione *user* dedicata agli include.

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

Serve poi definire alcune variabili, il buffer dei messaggi che invieremo sulla porta seriale, una per memorizzare la lunghezza dei messaggi ed una il valore letto dal timer

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

Inviamo poi sulla seriale uno splash message ed avviamo il 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 */
```

A questo punto siamo pronti a leggere il valore del timer. Facciamo due letture consecutive separate da un ritardo noto di 50 ms, e dopo aver misurato per differenza il tempo trascorso, lo inviamo sulla porta seriale.

Infine aspettiamo ancora 1s in modo da avere misure distanziate.

```
/* 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 */
```

Compilare e scaricare il programma sulla scheda Nucleo.

Per visualizzare i messaggi inviati dalla scheda Nucleo sarà necessario utilizzare un client, aprire la connessione sulla porta assegnata alla Nucleo dal sistema operativo, selezionando 115200 bps come velocità di connessione.

I messaggi visualizzati riporteranno il tempo di attesa di 50ms espresso in microsecondi.

# Capitolo 9 PWM

In questo capitolo approfondiremo l'uso dei segnali PWM (Pulse-Width Modulation) sul microcontrollore STM32, illustrando i concetti fondamentali e guidandoti nella configurazione pratica del progetto.

## Obiettivi del Progetto

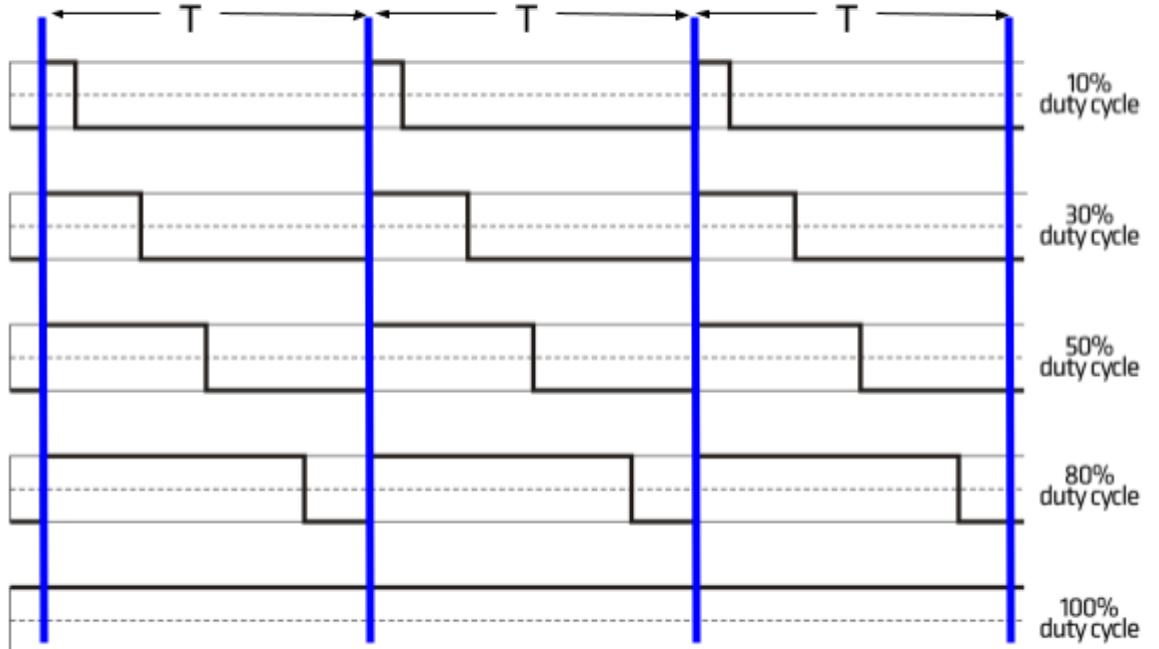
L'obiettivo principale di questo progetto è farti sperimentare e comprendere il funzionamento del PWM in ambiente STM32. In particolare, imparerai a:

- **Generare segnali PWM:** Configurare i canali di un timer per produrre uscite PWM con frequenza e duty-cycle variabili.
- **Regolare il duty-cycle:** Modificare la larghezza dell'impulso per controllare intensità luminosa di un ledi.
- **Gestire modalità avanzate:** Usare PWM in modalità center-aligned, generazione di uscite complementari con dead-time e gestione delle fault.

## Teoria di Base: Cos'è il PWM

La modulazione di larghezza d'impulso (PWM) è una tecnica per generare un segnale digitale periodico in cui la durata del livello “alto” (duty-cycle) è variabile, pur mantenendo costante la frequenza complessiva.

- **Frequenza PWM:** Determina il periodo  $T=1/f$  del segnale ed è definito da come vengono impostati il prescaler e il valore di Auto-Reload Register (ARR) del timer.
- **Duty-Cycle:** Indica la percentuale di tempo in cui il segnale rimane “alto” all'interno di ogni periodo, ed è controllato dal valore di confronto (CCR). Un duty-cycle del 50 % significa metà periodo “alto” e metà “basso”.



I principali registri coinvolti per la generazione di un segnale PWM sono:

- **Prescaler (PSC):** divide la frequenza del clock di base per rallentare il conteggio del timer, permettendo di ottenere frequenze PWM più basse o risoluzioni più fini.
- **Auto-Reload Register (ARR):** definisce il valore massimo del contatore; quando viene raggiunto, il timer si resetta e ricomincia da zero, stabilendo così il periodo del PWM.
- **Capture/Compare Register (CCRx):** contiene il valore di soglia per il duty-cycle; quando il contatore supera questo valore, l'uscita cambia stato (da “alto” a “basso” o viceversa).

e quindi per generare un segnale PWM dobbiamo agire opportunamente su questi registri.

Il primo passo consiste nell'impostare i registri **PSC** e **ARR** in modo da ottenere la frequenza desiderata:

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

Impostata la frequenza, dobbiamo definire il duty-cycle e lo facciamo impostando il **CCR** per definire quando il segnale deve tornare a zero.

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

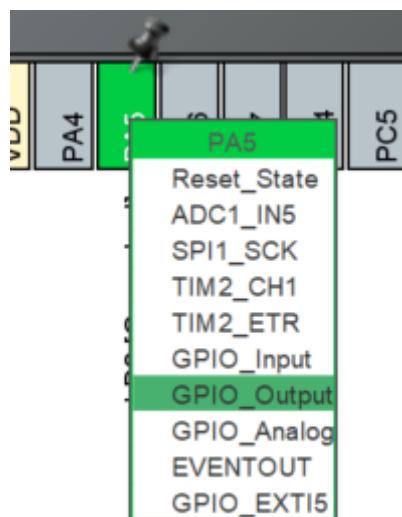
## Configurare il Progetto

Apri STM32CubeIDE, seleziona il workspace creato in precedenza e crea un progetto dal nome *PWM*. Se non ricordi come fare, puoi rivedere il capitolo precedente, la procedura di creazione e di configurazione di un progetto non cambia.

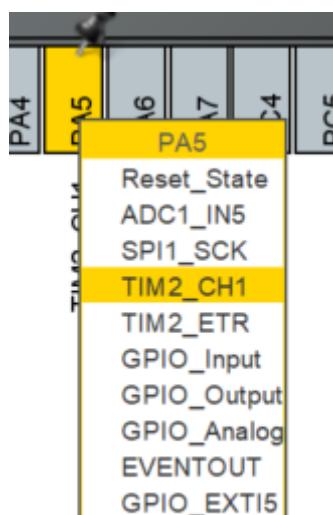
Con un doppio click sul file *pwm.ioc* accediamo alla configurazione del microcontrollore.

### Configurazione del pin A5

Il led LD2 è connesso al pin A5, pertanto dobbiamo configurare il timer a cui fa riferimento questo pin. Selezioniamo il pin A5 nel configuratore, che se abbiamo scelto in fase di creazione del progetto di procedere alla configurazione di default, risulterà inizializzato come *GPIO\_Output*.

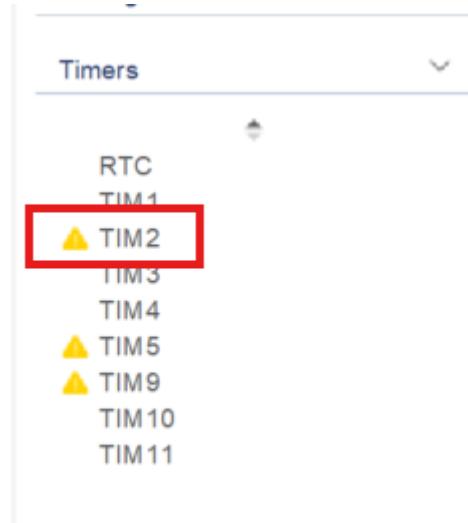


Selezioniamo per questo pin il canale 1 del timer 2



La selezione assume il colore giallo perché è necessario procedere con altri passi di configurazione al termine dei quali, se tutto è stato configurato correttamente assumerà il colore verde.

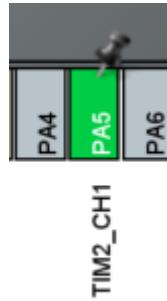
Nel menù *Timers* selezioniamo il TIM2



Nel pannello di configurazione *TIM2 Mode and Configuration* selezioniamo per il canale 1 la generazione di un segnale PWM e come *Clock Source* l'Internal Clock.

The screenshot shows the 'TIM2 Mode and Configuration' panel. On the left, there's a sidebar with categories like System Core, Analog, and Timers, with TIM2 selected. The main panel has a 'Mode' tab. Under 'Mode', there are several configuration fields: 'Slave Mode' (Disable), 'Trigger Source' (Disable), 'Clock Source' (Internal Clock), 'Channel1' (PWM Generation CH1), 'Channel2' (Disable), 'Channel3' (Disable), 'Channel4' (Disable), 'Combined Channels' (Disable), and three checkboxes at the bottom: 'Use ETR as Clearing Source', 'XOR activation', and 'One Pulse Mode'. The 'Clock Source' and 'Channel1' fields are highlighted with red boxes.

Questa configurazione è sufficiente a rendere il pin PA5 di colore verde, ma la definizione del segnale *pwm* desiderato, non è ancora completa.

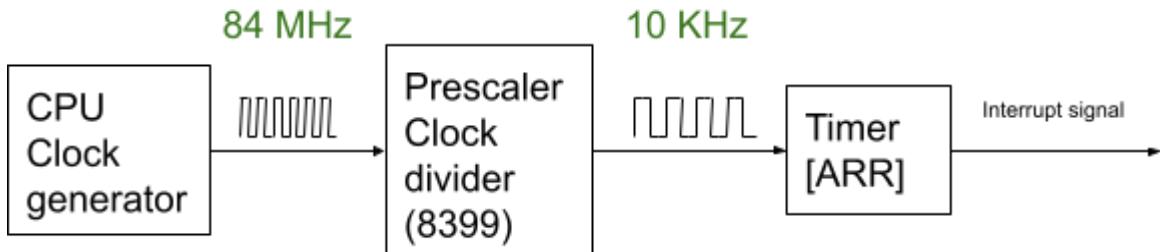


## Esempio 1

Come primo caso, vogliamo fare accendere il led della scheda Nucleo per 1 secondo e spegnerlo per 1 secondo.

Per ottenere questo risultato, abbiamo bisogno di un segnale che abbia un periodo di 2 secondi con un duty cycle del 50%.

Per il *Prescaler* inseriamo il valore 8399 per portare il segnale di clock (84 MHz) a 10 KHz.



$PSC+1 = 8400$ ; abbassa 84 MHz a 10 kHz di conteggio. Ricordiamo infatti che il prescaler è un registro il cui valore parte da zero. Inserire il valore 8399 equivale ad inserire un fattore di scale  $8399+1=8400$ .

Quindi in uscita dal prescaler avremo un segnale di clock di frequenza:

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

Il segnale ottenuto dal prescaler è quindi un segnale che ha un periodo di:

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

Per calcolare il valore da inserire nel registro *ARR*, ossia della durata del periodo di 2 secondi, dobbiamo capire quanti impulsi del segnale a 10 KHz dobbiamo contare.

Vogliamo che il segnale sia alto per 1 secondo e basso per il restante secondo del periodo.

Dobbiamo quindi contare un numero di impulsi del segnale a 10 KHz per formare 2 secondi.

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

e dunque

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

Non resta che definire il *Ton* ossia la durata dell'impulso all'interno del periodo. Lo facciamo andando ad inserire il valore 9999 nel campo *Pulse*.

Infatti

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

da cui:

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

In CubelDE, nel pannello dei *Parameter settings*, il valore assegnato al registro CCR è indicato con *Pulse*.

La seguente figura riporta il pannello *Parameter Settings* con i valori da assegnare, lasciando gli altri al loro valore di default.

Per migliorare la leggibilità, è buona norma inserire il valore desiderato nei registri sottraendo uno per tener conto che i valori partono da zero.

A questo punto possiamo generare il codice salvando il progetto ed accettando la generazione del codice.

Non resta che aggiungere il codice per avviare la generazione del segnale.

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

Compilare e scaricare nella Nucleo come già fatto in precedenza. Vedremo il led verde lampeggiare.

## Esempio 2:

Adesso vogliamo ottenere un effetto *fade-in* sul led, vogliamo cioè che questo partendo da spento si illuminì progressivamente per poi ricominciare il ciclo.

Per ottenere l'effetto *fade-in*, dobbiamo riprendere il file *pwm.ioc* ed effettuare delle modifiche.

Vogliamo portare la frequenza della pwm ad 1 KHz partendo da una frequenza a valle del prescaler di 1 MHz

Nel *prescaler* inseriamo il valore *84-1*. Infatti:

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

quindi

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

Per quanto riguarda il valore da inserire nel *ARR*, partiamo dalla frequenza di 1 KHz che vogliamo abbia il segnale pwm.

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

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

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

$$ARR = \frac{0,001}{10^{-6}} - 1 = 10^3 - 1 = 999$$

Dunque il valore da inserire nel *ARR* è 999.

Per quanto riguarda il *CRR* (*Pulse*) lo lasciamo a 0 e lo modificheremo opportunamente da codice.

Reset Configuration

<input checked="" type="checkbox"/> NVIC Settings	<input checked="" type="checkbox"/> DMA Settings	<input checked="" type="checkbox"/> GPIO Settings
<input checked="" type="checkbox"/> Parameter Settings	<input checked="" type="checkbox"/> User Constants	

Configure the below parameters :

Search (Ctrl+F) (F) (B) (I)

- ▽ Counter Settings
  - Prescaler (PSC - 16 bits value) 84-1
  - Counter Mode Up
  - Counter Period (AutoReload ...) 1000-1
  - Internal Clock Division (CKD) No Division
  - auto-reload preload Disable
- ▽ Trigger Output (TRGO) Parameters
  - Master/Slave Mode (MSM bit) Disable (Trigger input effect not delayed)
  - Trigger Event Selection Reset (UG bit from TIMx\_EGR)
- ▽ PWM Generation Channel 1
  - Mode PWM mode 1
  - Pulse (32 bits value) 0
  - Output compare preload **Enable**
  - Fast Mode Disable
  - CH Polarity High

```
/* 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 */
}
```

Il codice avvia la generazione del segnale *pwm*, e legge il valore del periodo dal registro *ARR* per assegnarlo alla variabile *max Pulse*. Il ciclo *for* assegna al *ton* ossia a *pulse* un valore che parte da 0 ed arriva fino alla durata del periodo, ossia al 100% del *dutycycle*.

A questo punto possiamo generare il codice, compilare e scaricare nella Nucleo come già fatto in precedenza. Vedremo la luminosità del led verde aumentare da zero fino al suo valore massimo.