

# Laboratorio di Misure a Microcontrollore

## --- MAPI FOR DUMMIES ---

LICCARDO – SCHIANO LO MORIELLO  
Fabrizio Guillaro

## Indice

<b>1. Introduzione</b>	<b>3</b>
1.1    Specifiche della scheda	3
1.2    Opzioni del progetto	4
1.3    Visualizzazione dei registri	5
1.4    Reset della scheda	5
<b>2. Periferiche GPIO</b>	<b>6</b>
2.1    Descrizione generale	6
2.2    Registri utili delle periferiche (GPIOx)	6
2.3 <b>Progetto 1:</b> accensione LED blu	8
2.4 <b>Progetto 2:</b> accensione di più LED	11
2.5 <b>Progetto 3:</b> accensione LED con pulsante USER	12
<b>3. Librerie della scheda STM</b>	<b>15</b>
3.1    Struttura GPIO_Type	15
3.2    Libreria stm32f30x.h	16
3.3 <b>Progetto 4:</b> progetto 3 con uso della libreria	18
<b>4. Contatore binario</b>	<b>19</b>
4.1 <b>Progetto 5:</b> realizzazione di un contatore binario (8 bit)	19
<b>5. Timer</b>	<b>20</b>
5.1    Timer general-purpose	20
5.2    Registri utili dei timer	21
5.3 <b>Progetto 6:</b> LED lampeggianti	22
5.4 <b>Progetto 7:</b> misurazione intervallo di tempo	25
<b>6. Interruzioni (NVIC)</b>	<b>28</b>
6.1    Controllore delle interruzioni (NVIC)	28
6.2    File startup_stm32f303xc.s	29
6.3    File system_stm32f30x.c	30
6.4 <b>Progetto 8:</b> LED lampeggianti con uso delle interruzioni	31

<b>7. Convertitore A/D</b>	<b>33</b>
7.1    Descrizione generale	33
7.2    Registri utili dell'ADC	36
7.3 <b>Progetto 9:</b> conversione tensione del pulsante USER	40
<b>8. Convertitore D/A</b>	<b>43</b>
8.1    Descrizione generale	43
8.2    Registri utili del DAC	45
8.3 <b>Progetto 10:</b> DAC – ADC in serie	47
8.4 <b>Progetto 11:</b> generazione sinusoidale	50
<b>9. Interruzioni (EXTI)</b>	<b>55</b>
9.1    Interruzioni delle GPIO	55
9.2    Registri utili delle interruzioni EXTI	56
9.3 <b>Progetto 12:</b> conversione alla pressione di USER	58
<b>10. DMA</b>	<b>61</b>
10.1   Descrizione generale	61
10.2   Registri utili del DMA	62
10.3 <b>Progetto 13:</b> generazione sinusoidale con DMA	64
<b>11. Accelerometro</b>	<b>70</b>
11.1   Interfaccia I <sup>2</sup> C	70
11.2   Registri utili dell'I <sup>2</sup> C	74
11.3   Accelerometro	76
11.4   Registri utili dell'accelerometro	76
11.5 <b>Progetto 14:</b> lettura accelerazione	77

## Premessa

M4D (MAPI for Dummies) è stato scritto seguendo le lezioni della professoressa Liccardo (2016/17), per cui gli esercizi potrebbero leggermente variare rispetto a quelli proposti dal prof. Schiano o a quelli proposti durante altri anni accademici.

Lo svolgimento di ciascun esercizio non è, ovviamente, l'unico possibile o il più efficace, ma semplicemente quello che ho realizzato io (Fabrizio) insieme al mio team (Lino e Corrado). Tutto il codice è però testato e funzionante, quindi se non funziona è probabile che tu abbia commesso qualche errore (magari hai scordato qualche opzione del progetto o devi resettare la scheda).

Consiglio di stamparlo a colori per migliore leggibilità o comunque tenere la versione pdf, che è dotata di indice (segnalibri).

Perdonate eventuali errori di battitura o di distrazione e tristi freddure.

Enjoy.

# 1. Introduzione

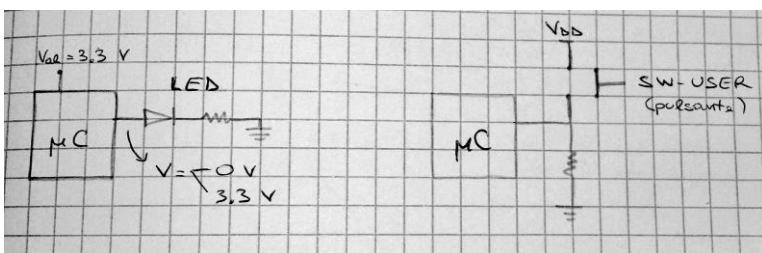
## 1.1 Specifiche della scheda

Scheda: **STM32F3 Discovery**.

La scheda è dotata di:

- Microcontrollore **STM32F303VC** (32 bit)
- Giroscopio triassiale
- Magnetometro triassiale
- Accelerometro triassiale
- Oscillatore – fornisce il segnale di clock (8 MHz)
- USB ST-LINK
- USB USER
- Pulsante RESET
- Pulsante USER

Il microcontrollore ha un processore Cortex-M4F con 48 KB di SRAM e 256 KB di memoria flash.



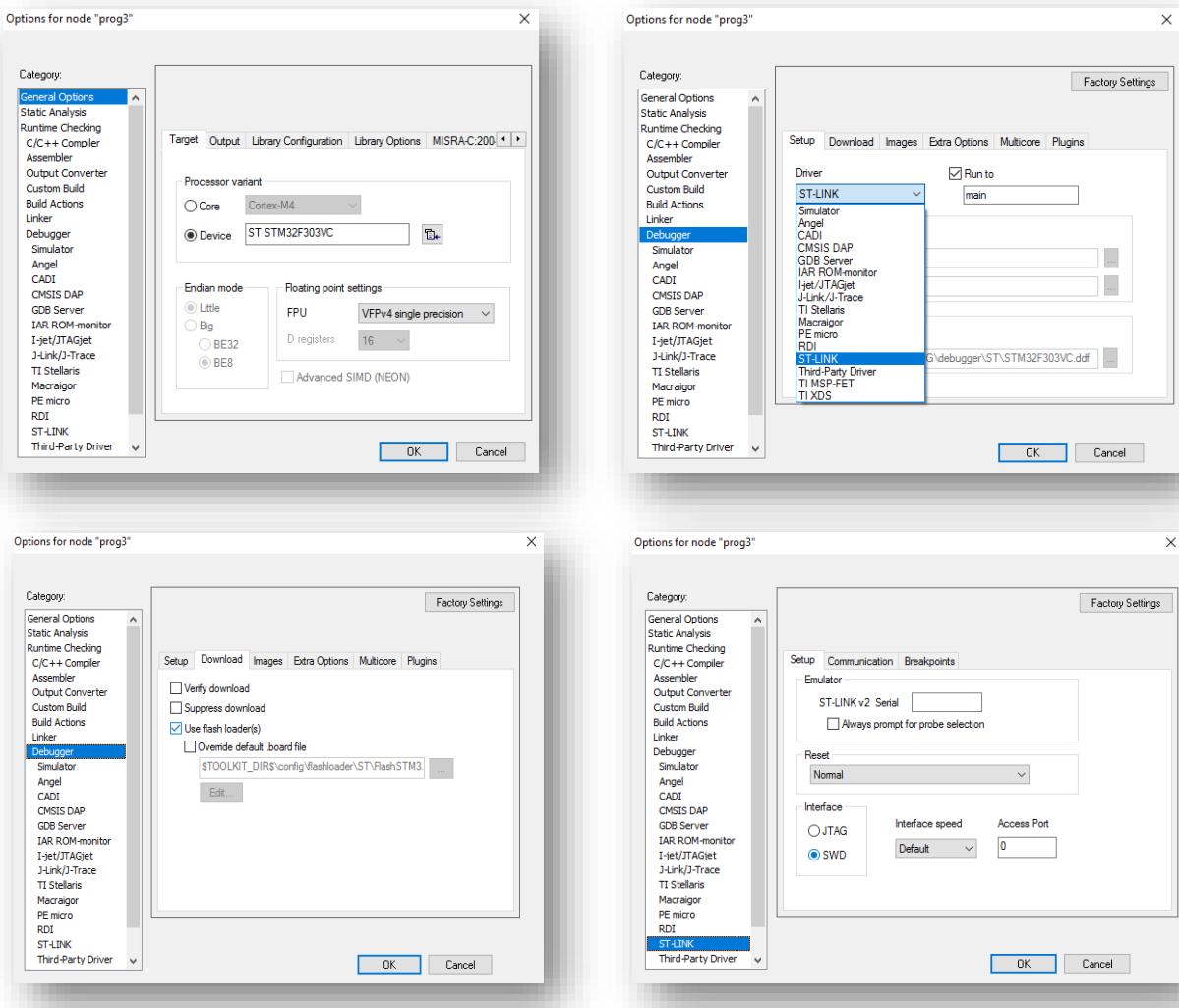
Schema dei LED e del pulsante USER

## 1.2 Opzioni del progetto

Ambiente di sviluppo (IDE): **IAR Embedded Workbench (ARM)**.

**Opzioni** preliminari da impostare ad OGNI progetto su IAR (click destro sul progetto > Options):

- General Options > Target > Device > STM32F303VC
- Debugger > Setup > Driver > ST-LINK
- Debugger > Download > Use flash loader(s)
- ST-LINK > Setup > SWD



## 1.3 Visualizzazione dei registri

Per visualizzare i registri della scheda durante il debugging, basta cliccare su

- View > Registers

mentre è attivo il debugging.

GPIOE		<find register>
+ MODER	= 0x00000000	
+ OTYPER	= 0x00000000	
+ OSPEEDR	= 0x00000000	
+ PUPDR	= 0x00000000	
+ IDR	= 0x00000000	
+ ODR	= 0x00000000	
+ BSRR	= WWWW	
+ LCKR	= 0x00000000	
+ AFRL	= 0x00000000	
+ AFRH	= 0x00000000	
+ BRR	= WWWW	

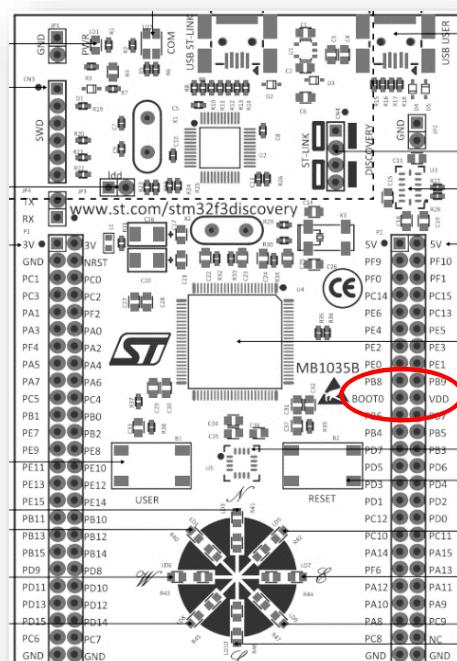
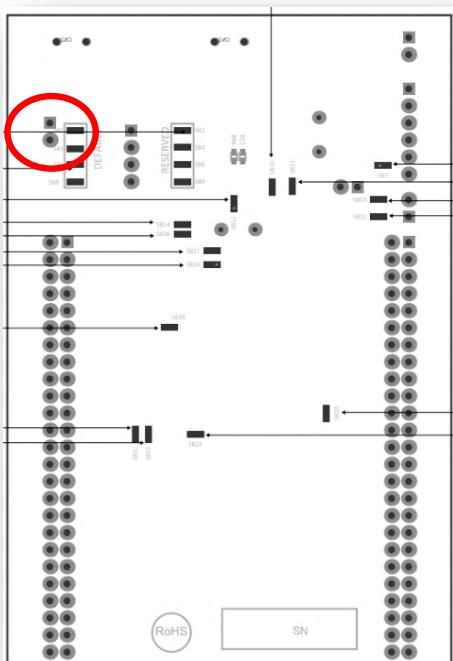
## 1.4 Reset della scheda

Per effettuare un reset del dispositivo e ripristinare i dati di fabbrica, occorre:

- Scollegare la scheda
- Rimuovere il jumper JP2 dal retro della scheda
- Collegare i pin BOOT0 e VDD a destra della scheda con il jumper
- Collegare la scheda
- Eseguire un programma semplice (ad esempio un main vuoto)
- Scollegare la scheda
- Mettere al suo posto il jumper

Mettendo a corto circuito BOOT0 e VDD, la scheda ignora il programma inserito dall'utente ed esegue invece il programma di boot, che resetta il dispositivo.

**Nota:** penso vada bene anche il jumper JP1, che è quello in alto a destra



## 2. Periferiche GPIO

### 2.1 Descrizione generale

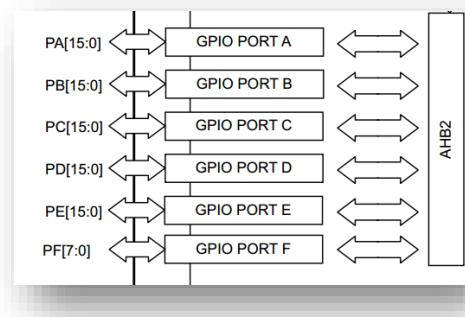
La scheda è dotata di 6 porte GPIO (*General Purpose Input/Output*), da A ad F.

Ogni porta è dotata di un certo numero di linee, in particolare le porte A ... E hanno 16 linee, mentre la porta F ne ha solo 8.

Alcune porte sono collegate agli 8 LED della scheda, altre corrispondono invece ai canali di ingresso e uscita del convertitore D/A e del convertitore A/D, altre ancora ai pulsanti USER e RESET e così via.

Ogni linea può operare in modalità analogica o digitale e fungere da input o da output a seconda di come sono state impostate nel registro MODER.

In più, una linea può anche operare in modalità “Alternate Function”, che permette di sfruttare altri meccanismi messi a disposizione dalla STM.



### 2.2 Registri utili delle periferiche (GPIOx)

I registri più utili sono:

- **GPIOx\_MODER (Mode Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]	MODER14[1:0]	MODER13[1:0]	MODER12[1:0]	MODER11[1:0]	MODER10[1:0]	MODER9[1:0]	MODER8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]	MODER6[1:0]	MODER5[1:0]	MODER4[1:0]	MODER3[1:0]	MODER2[1:0]	MODER1[1:0]	MODER0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- MODERx
  - 00: linea x in modalità input
  - 01: linea x in modalità output
  - 10: linea x in modalità alternate-funcion
  - 00: linea x in modalità analogica

- **GPIOx\_IDR (Input Data Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- **IDRx**

- 0: legge uscita bassa sulla linea x
- 1: legge uscita alta sulla linea x

- **GPIOx\_ODR (Output Data Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **ODRx**

- 0: imposta uscita bassa sulla linea x
- 1: imposta uscita alta sulla linea x

- **GPIOx\_AFRL e GPIOx\_AFRH (Alternate Function Low/High Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw												

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw												

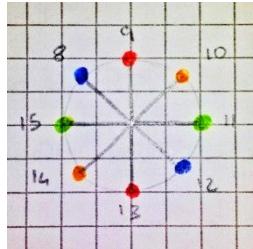
- **AFRL/AFRH**

- 0000: AF0
- 0001: AF1
- ...
- 1111: AF15

## 2.3 Progetto 1: accensione LED blu

I LED si trovano nella porta E, in particolare:

- PE8 Blu
- PE9 Rosso
- PE10 Arancio
- PE11 Verde
- PE12 Blu
- PE13 Rosso
- PE14 Arancio
- PE15 Verde



PE8	TIM1_CH1N	39			LD4/ BLUE	
PE9	TIM1_CH1	40			LD3/ RED	
PE10	TIM1_CH2N	41			LD5/ ORANGE	
PE11	TIM1_CH2	42			LD7/ GREEN	
PE12	TIM1_CH3N	43			LD9/ BLUE	
PE13	TIM1_CH3	44			LD10/ RED	
PE14	TIM1_CH4_BKIN2	45			LD8/ ORANGE	
PE15	TIM1_BKIN, USART3_RX	46			LD6/ GREEN	

Per accendere il LED blu (PE8) bisogna:

1. Attivare il clock della porta E
2. Abilitare la linea 8 come output
3. Accendere la PE8

### Attivare il clock della porta E:

Le porte GPIO sono collegate al bus AHB2. Per poter lavorare sulla periferica E bisogna accedere al registro **RCC\_AHBENR** (*Reset and Clock Control – AHB peripheral clock Enable Register*).

AHB1	AHB peripheral clock enable register (RCC_AHBENR)		
	Address offset: 0x14		
0x4002 2000 - 0x4002 23FF	1 K	Flash interface	
0x4002 1400 - 0x4002 1FFF	3 K	Reserved	
0x4002 1000 - 0x4002 13FF	1 K	RCC	
0x4002 0800 - 0x4002 0FFF	2 K	Reserved	

L'indirizzo dei registri RCC è **0x4002 1000**, a cui va aggiunto l'offset **0x14** dell'AHBENR.

→ L'indirizzo di **RCC\_AHBENR** è **0x4002 1014**

Bisogna ora alzare il bit "IOPE EN" (*I/O Port E clock Enable*) di questo registro per abilitare la porta E e codificare in esadecimale.

→ Il valore di **RCC\_AHBENR** è **0x0020 0000**, analogo a **(1 << 21)**, cioè 1 traslato di 21 posti

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	ADC34EN	ADC12EN	Res	Res	Res	TSCEN	Res	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res
		rw	rw				rw		rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	CRC EN	Res	FLITF EN	Res	SRAM EN	DMA2E N	DMA1 EN
									rw		rw		rw		rw

## Abilitare la linea 8 come output:

Per impostare la linea 8 come output bisogna accedere al registro **GPIOE\_MODER** (*GPIO port E Mode Register*).

AHB2	0x4800 1400 - 0x4800 17FF	1 K	GPIOF
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 K	GPIOD
	0x4800 0800 - 0x4800 0BFF	1 K	GPIOC
	0x4800 0400 - 0x4800 07FF	1 K	GPIOB
	0x4800 0000 - 0x4800 03FF	1 K	GPIOA

**GPIO port mode register (GPIOx\_MODER) (x = A..F)**  
 Address offset: 0x00

L'indirizzo dei registri GPIOE è **0x4800 1000**, a cui va aggiunto l'offset **0x00** del MODER.

→ L'indirizzo di **GPIOE\_MODER** è **0x4800 1000**

Bisogna quindi impostare i bit "MODER8" a 01 per configurare la linea 8 come output.

→ Il valore di **GPIOE\_MODER** è **0x0001 0000**, analogo a **(1 << 16)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]	MODER14[1:0]	MODER13[1:0]	MODER12[1:0]	MODER11[1:0]	MODER10[1:0]	MODER9[1:0]	MODER8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]	MODER6[1:0]	MODER5[1:0]	MODER4[1:0]	MODER3[1:0]	MODER2[1:0]	MODER1[1:0]	MODER0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

## Accendere la PE8:

Per accendere il LED blu collegato a PE8 bisogna accedere al registro **GPIOE\_ODR** (*GPIO port E Output Data Register*)

<b>GPIO port output data register (GPIOx_ODR) (x = A..F)</b>															
Address offset: 0x14															

L'offset del ODR è **0x14**.

→ L'indirizzo di **GPIOE\_ODR** è **0x4800 1014**

Bisogna ora alzare il bit "ODR8" per accendere il LED.

→ Il valore di **GPIOE\_ODR** è **0x0000 0100**, analogo a **(1 << 8)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

**Nota:** per alzare un solo bit si può anche usare il registro **GPIOE\_BSRR** (*Bit Set/Reset Register*)

```
#define RCC_AHBENR 0x40021014 //attivazione periferiche
#define GPIOE_MODER 0x48001000 //abilitare LED - PE8...PE15
#define GPIOE_ODR 0x48001014 //accendere LED

unsigned int* p;

void main(){
    //attivazione PE
    p = (unsigned int*) RCC_AHBENR;
    *p = (1<<21);
    //abilitazione LED blu (PE8) come output
    p = (unsigned int*) GPIOE_MODER;
    *p = (1<<16);
    //accensione LED blu
    p = (unsigned int*)GPIOE_ODR;
    *p = (1<<8);

    while(1);
}
```

Accensione LED blu

## 2.4 Progetto 2: accensione di più LED

È simile all'esercizio precedente, ma i contenuti dei registri MODER e ODR cambiano.

Se ad esempio vogliamo accendere i due LED della PE8 (blu) e della PE11 (verde), dobbiamo assicurarsi che il MODER imposti entrambe le linee 8 e 11 come output e che l'ODR accenda effettivamente quei due LED.

- Il valore di GPIOE\_MODER è **0x0041 0000**, analogo a  $(1 \ll 16) / (1 \ll 22)$
- Il valore di GPIOE\_ODR è **0x0000 0900**, analogo a  $(1 \ll 8) / (1 \ll 11)$

**Nota:** Il simbolo ‘|’ è un ‘OR’ ed è necessario per alzare entrambi i bit (16 e 22, nell'esempio del MODER). Ad esempio:

*p = (1 << 16);	*p = (1 << 16);
*p = (1 << 22); //sovrascrive il bit 16	*p  = (1 << 22); //esegue l'OR, quindi non azzera i bit già alti

```
#define RCC_AHBENR 0x40021014 //attivazione periferiche
#define GPIOE_MODER 0x48001000 //abilitare LED - PE8...PE15
#define GPIOE_ODR 0x48001014 //accendere LED

unsigned int* p;

void main() {
    //attivazione PE
    p = (unsigned int*) RCC_AHBENR;
    *p = (1<<21);
    //abilitazione LED blu (PE8) e LED verde (PE11) come output
    p = (unsigned int*) GPIOE_MODER;
    *p = (1<<16) | (1<<22);
    //accensione LED blu e verde
    p = (unsigned int*)GPIOE_ODR;
    *p = (1<<8) | (1<<11);

    while(1);
}
```

Accensione LED blu e verde

## 2.5 Progetto 3: accensione LED con pulsante USER

Il pulsante USER è collegato alla linea 0 della porta A.

Quindi, se vogliamo accendere gli 8 LED alla pressione di USER, l'AHBENR deve abilitare sia il clock della porta E che della porta A.

- Il valore di RCC\_AHBENR è  $(1 \ll 21) / (1 \ll 17)$

Main function	Alternate functions	MCU pin		Board function					
		LQFP100 pin num.	LSM303DLHC	L3GD20	Pushbutton	LED	SWD	USB	OSC
BOOT0		94							
NRST		14			RESET			NRST	
PA0	TIM2_CH1_ETR, G1_IO1, USART2_CTS, COMP1_OUT, TIM8_BKIN, TIM8_ETR	23			USER				

Per accendere tutti i LED bisogna rendere output (01) tutte le linee a cui sono collegati i LED, cioè da PE15 a PE8.

- Il valore di GPIOE\_MODER è 0x5555 0000

Per impostare PA0 come input, accediamo al registro **GPIOA\_MODER**.

AHB2	0x4800 1400 - 0x4800 17FF	1 K	GPIOF
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 K	GPIOD
	0x4800 0800 - 0x4800 0BFF	1 K	GPIOC
	0x4800 0400 - 0x4800 07FF	1 K	GPIOB
	0x4800 0000 - 0x4800 03FF	1 K	GPIOA

**GPIO port mode register (GPIOx\_MODER) (x = A..F)**  
 Address offset: 0x00

- L'indirizzo di GPIOA\_MODER è 0x4800 0000

Gli ultimi due bit del GPIOA\_MODER sono "MODER0" e vanno messi a 00 per configurare la linea 0 come input. Tuttavia vogliamo solo impostare questi 2 bit a 0, senza modificare gli altri presenti nel registro; per realizzare ciò, si utilizza l'operatore '&', che rappresenta un 'AND'.

- Il valore di GPIOA\_MODER è &= 0xFFFF FFFC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw										

**Nota importante:** le modifiche ai registri della porta A possono essere fatali! È per questo motivo che in GPIOA\_MODER ci preoccupiamo di non alterare gli altri bit. In caso di malfunzionamenti del dispositivo, effettuare il RESET come mostrato nel paragrafo 1.4

Adesso vogliamo che i LED si accendano solo se premiamo il bottone:

- ❖ SE il bottone è premuto ALLORA accendi i led, ALTRIMENTI spegnili.

Per vedere se USER è premuto interroghiamo il registro **GPIOA\_IDR** (*Input Data Register*). L'IDR ha un'offset di *0x10*.

→ L'indirizzo di **GPIOA\_IDR** è  
**0x4800 0010**

**GPIO port input data register (GPIOx\_IDR) (x = A..F)**  
Address offset: 0x10

Poiché USER è collegato a PA0, basta verificare che l'ultimo bit è alto. Per non dar conto ai bit precedenti, il cui valore non ci interessa, si usa l'operatore AND e si pone il valore del registro uguale a 1.

**Nota:** Se non si utilizza AND e si pone direttamente uguale a 1 si commette un errore, perché noi vogliamo solo sapere se l'ultimo bit sia 1, non vogliamo che tutti i precedenti siano necessariamente a 0.

→ La condizione da verificare è che **GPIOA\_MODER & 1** sia vero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Non ci resta quindi che accendere i LED quando la condizione è verificata:

→ Il valore di **GPIOA\_ODR** è **0x0000 FF00**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```

#define RCC_AHBENR 0x40021014 //attivazione periferiche
#define GPIOA_MODER 0x48000000 //pulsante USER - PA0
#define GPIOA_IDR 0x48000010 //lettura input
#define GPIOE_MODER 0x48001000 //abilitare LED - PE8...PE15
#define GPIOE_ODR 0x48001014 //accendere LED

unsigned int* p;

void main() {
    //attivazione PA e PE
    p = (unsigned int*)RCC_AHBENR;
    *p = (1<<21) | (1<<17);
    //abilitazione LED (PE8-PE15) come output
    p = (unsigned int*)GPIOE_MODER;
    *p = 0x55550000;
    //abilitazione PA0 come input
    p = (unsigned int*)GPIOA_MODER;
    *p &= 0xFFFFFFFFFC;

    while(1) {
        //lettura input
        p = (unsigned int*)GPIOA_IDR;

        if((*p&1) == 1){
            p = (unsigned int*)GPIOE_ODR;
            *p = 0x0000FF00;
        }
        else{
            p = (unsigned int*)GPIOE_ODR;
            *p = 0x00000000;
        }
    }
}

```

Accensione LED con pulsante USER

# 3. Librerie della scheda STM

## 3.1 Struttura GPIO\_Type

Ogni classe di registri ha un indirizzo base (*Base Address*). In particolare gli indirizzi dei GPIO sono:

- GPIOA - 0X4800 0000
- GPIOB - 0X4800 0400
- GPIOC - 0X4800 0800
- GPIOD - 0X4800 0C00
- GPIOE - 0X4800 1000
- GPIOF - 0X4800 1400

Inoltre, tutte le GPIOx hanno gli stessi tipi di registri con i medesimi offset:

- MODER 0x00
- OTYPER 0x04
- OSPEEDR 0x08
- PUPDR 0x0C
- IDR 0x10
- ODR 0x14
- BSRR 0x18
- LCKR 0x1C
- AFRL 0x20
- AFRH 0x24
- BRR 0x28

Si può allora pensare di organizzare i GPIOx in una struttura dati.

L'ordine secondo cui si presentano i registri all'interno della struct deve seguire l'ordine degli offset.

In questo modo è possibile realizzare programmi molto più puliti e leggibili, in quanto indirizzi e offset sono nascosti nella struttura dati.

Ovviamente lo stesso discorso vale per qualsiasi altra classe di registri, come ad esempio gli RCC.

```
//definizione del tipo GPIO

typedef{
    struct{
        unsigned int MODER;
        unsigned int OTYPER;
        unsigned int OSPEEDR;
        unsigned int PUPDR;
        unsigned int IDR;
        unsigned int ODR;
        unsigned int BSRR;
        unsigned int LCKR;
        unsigned int AFRL;
        unsigned int AFRH;
        unsigned int BRR;
    };
} GPIO_Type;

//indirizzi base

#define GPIOA (GPIO_Type*) 0x48000000
#define GPIOB (GPIO_Type*) 0x48000400
#define GPIOC (GPIO_Type*) 0x48000800
#define GPIOD (GPIO_Type*) 0x48000C00
#define GPIOE (GPIO_Type*) 0x48001000
#define GPIOF (GPIO_Type*) 0x48001400

//esempio di utilizzo

GPIOE->ODR |= (1 << 8);
```

## 3.2 Libreria stm32f30x.h

Fortunatamente, non siamo noi a doverci occupare di realizzare queste strutture dati: la STM ha realizzato delle librerie utili che ci semplificano il lavoro. La libreria che dovremo includere nei nostri programmi si chiama “**stm32f30x.h**” e contiene, tra le altre cose, anche delle strutture dati simili a quelle progettate nel paragrafo precedente.

Per esempio, la struttura vista prima è scritta così:

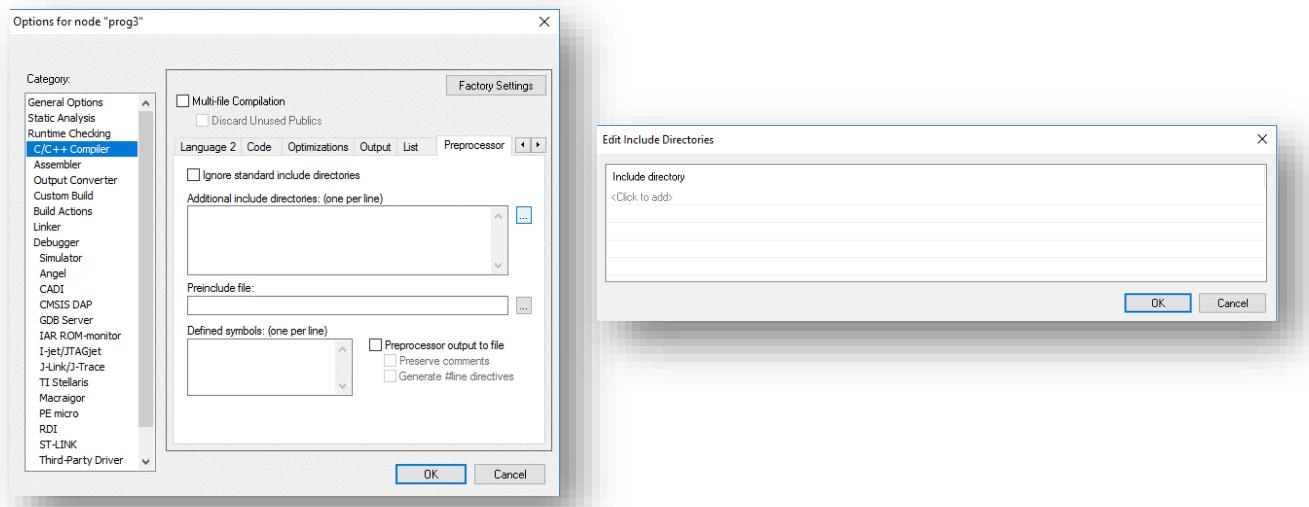
```
typedef struct
{
    __IO uint32_t MODER;           /*!< GPIO port mode register,
    __IO uint16_t OTYPER;          /*!< GPIO port output type register,
    uint16_t RESERVED0;
    __IO uint32_t OSPEEDR;         /*!< GPIO port output speed register,
    __IO uint32_t PUPDR;           /*!< GPIO port pull-up/pull-down register,
    __IO uint16_t IDR;             /*!< GPIO port input data register,
    uint16_t RESERVED1;
    __IO uint16_t ODR;             /*!< GPIO port output data register,
    uint16_t RESERVED2;
    __IO uint32_t BSRR;            /*!< GPIO port bit set/reset registerBSRR,
    __IO uint32_t LCKR;             /*!< GPIO port configuration lock register,
    __IO uint32_t AFR[2];           /*!< GPIO alternate function low register,
    __IO uint16_t BRR;              /*!< GPIO bit reset register,
    uint16_t RESERVED3;
}GPIO_TypeDef;
```

	Address offset: 0x00 */ Address offset: 0x04 */ 0x06 */ Address offset: 0x08 */ Address offset: 0x0C */ Address offset: 0x10 */ 0x12 */ Address offset: 0x14 */ 0x16 */ Address offset: 0x18 */ Address offset: 0x1C */ Address offset: 0x20-0x24 */ Address offset: 0x28 */ 0x2A */
--	---

```
/*!< AHB2 peripherals */
#define GPIOA_BASE      (AHB2PERIPH_BASE + 0x0000)
#define GPIOB_BASE      (AHB2PERIPH_BASE + 0x0400)
#define GPIOC_BASE      (AHB2PERIPH_BASE + 0x0800)
#define GPIOD_BASE      (AHB2PERIPH_BASE + 0x0C00)
#define GPIOE_BASE      (AHB2PERIPH_BASE + 0x1000)
#define GPIOF_BASE      (AHB2PERIPH_BASE + 0x1400)
#define GPIOG_BASE      (AHB2PERIPH_BASE + 0x00001800)
#define GPIOH_BASE      (AHB2PERIPH_BASE + 0x00001C00)
```

**Nota:** per poter includere la libreria senza ogni volta copiare l'header file nella directory del progetto, occorre andare nelle opzioni di progetto e aggiungere la directory delle librerie STM:

- C/C++ Compiler > Preprocessor > Additional include directories > ... > Click to add



Con questo metodo, è possibile accedere ai registri molto più facilmente. Se per esempio cercassimo di accedere all'RCC\_AHBENR, non dovremo più andare a cercare l'indirizzo e l'offset del registro, ma semplicemente si scrive **RCC->AHBENR**.

Aprendo l'header **stm32f30x.h** possiamo trovare anche delle etichette per ciascun bit di ciascun registro, cosa che rende molto più leggibile il nostro programma e ne velocizza notevolmente la realizzazione.

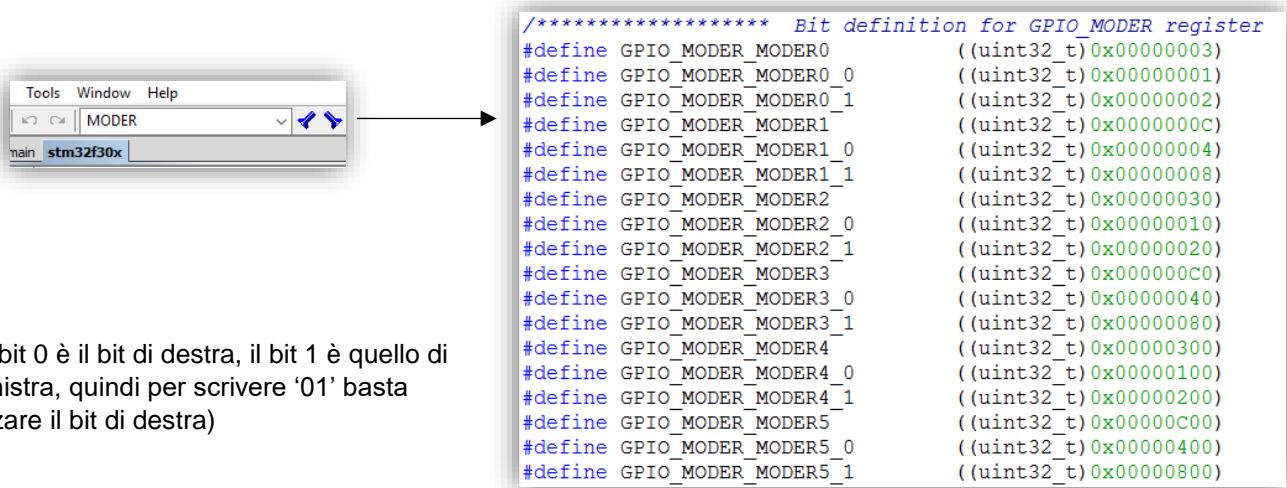
Se ad esempio volessimo impostare nel GPIOA\_MODER i bit MODER5 come output, anziché cercare nel Reference Manual il registro e la posizione del bit, basta cercare "MODER" nel file (per tenerlo a portata di mano conviene aggiungerlo al progetto).

In questo modo, al posto di scrivere:

```
GPIOA->MODER |= (1<<10);
```

si può scrivere:

```
GPIOA->MODER |= GPIO_MODER_MODER5_0;
```



(il bit 0 è il bit di destra, il bit 1 è quello di sinistra, quindi per scrivere '01' basta alzare il bit di destra)

Per i più nabbi esistono anche alcune "funzioni", tra cui SET\_BIT e CLEAR\_BIT.

In realtà non sono vere e proprie funzioni, ma delle semplici definizioni macro. Basta leggere il codice per capire che sintassi sostituiscono:

```
#define SET_BIT(REG, BIT) ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT) ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT) ((REG) & (BIT))
#define CLEAR_REG(REG) ((REG) = (0x0))
#define WRITE_REG(REG, VAL) ((REG) = (VAL))
#define READ_REG(REG) ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK) WRITE_REG((REG), (((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))
```

Per cui, facendo riferimento all'esempio di prima, si può scrivere:

```
SET_BIT ( GPIOA->MODER, GPIO_MODER_MODER5_0 );
```

### 3.3 Progetto 4: progetto 3 con uso della libreria

Mettendo a confronto i due codici sorgenti, ci si rende conto dell'evidente miglioria della leggibilità del programma. Inoltre sono anche più semplici ed immediati i ragionamenti per la realizzazione dello stesso.

```
#define RCC_AHBENR 0x40021014 //attivazione periferiche
#define GPIOA_MODER 0x48000000 //pulsante USER - PA0
#define GPIOA_IDR 0x48000010 //lettura input
#define GPIOE_MODER 0x48001000 //abilitare LED - PE8...PE15
#define GPIOE_ODR 0x48001014 //accendere LED

unsigned int* p;

void main() {
    //attivazione PA e PE
    p = (unsigned int*)RCC_AHBENR;
    *p = (1<<21)|(1<<17);
    //abilitazione LED (PE8-PE15) come output
    p = (unsigned int*)GPIOE_MODER;
    *p = 0x55550000;
    //abilitazione PA0 come input
    p = (unsigned int*)GPIOA_MODER;
    *p &= 0xFFFFFFFFC;

    while(1) {
        //lettura input
        p = (unsigned int*)GPIOA_IDR;

        if((*p&1) == 1){
            p = (unsigned int*)GPIOE_ODR;
            *p = 0x0000FF00;
        }
        else{
            p = (unsigned int*)GPIOE_ODR;
            *p = 0x00000000;
        }
    }
}
```

```
#include <stm32f30x.h>

void main() {

    RCC->AHBENR |= RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOEEN;      //attivazione PA e PE
    GPIOE->MODER = 0x55550000;                                //abilitazione LED (PE8-PE15) come output
    GPIOA->MODER &= ~GPIO_MODER_MODER0;    //abilitazione PA0 come input

    while(1) {
        if((GPIOA->IDR & GPIO_IDR_0) == GPIO_IDR_0)
            GPIOE->ODR = 0x0000FF00;                //accensione LED
        else
            GPIOE->ODR = 0;                      //spegnimento LED
    }
}
```

**Nota:** il simbolo ~ indica il negato.

# 4. Contatore binario

## 4.1 Progetto 5: realizzazione di un contatore binario (8 bit)

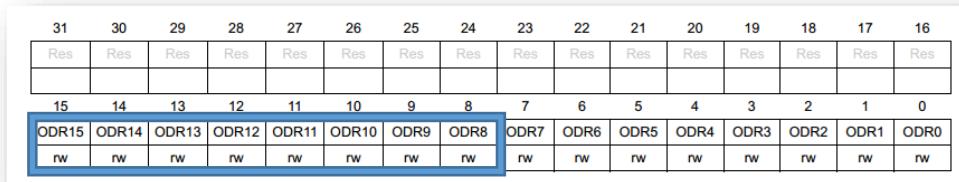
Per realizzare un contatore binario abbiamo ovviamente bisogno di una variabile contatore, il cui valore si incrementa di 1 ogni volta che il pulsante USER è premuto. Vogliamo inoltre che il valore sia visualizzato sul display, che nel nostro caso è rappresentato dagli 8 LED.

Tuttavia l'incremento deve avvenire solo al momento del rilascio del bottone, altrimenti il contatore aumenta di decine di migliaia di unità, a seconda di quanto a lungo il pulsante è premuto.

- SE il pulsante è premuto, ALLORA: aspetta che sia rilasciato, incrementa cont e visualizza il numero sul display

Per essere più precisi:

- SE il pulsante è premuto, ALLORA: FINCHÉ è premuto non fare niente, APPENA è rilasciato incrementa cont e visualizza il numero sul display



Nel registro ODR i LED, come già abbiamo visto, sono rappresentati dal terzo byte, cioè gli 8 bit dal 15° all' 8°. Perché la variabile cont possa essere “visualizzata”, basta traslare il valore di cont di 8 bit. Così:

- quando cont vale 1 (0000 0001), si accenderà il LED del bit 8
- quando cont vale 2 (0000 0010), si accenderà il LED del bit 9
- quando cont vale 3 (0000 0011), si accenderanno i LED dei bit 8 e 9
- ...

```
#include <stm32f30x.h>

int cont=0;

void main(){
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOEEN;      //attivazione PA e PE
    GPIOE->MODER = 0x55550000;                                     //abilitazione LED (PE8-PE15) come output
    GPIOA->MODER &= ~GPIO_MODER_MODERO;                           //abilitazione PA0 come input

    while(1){
        if((GPIOA->IDR & GPIO_IDR_0) == GPIO_IDR_0){           //se USER è premuto
            while((GPIOA->IDR & GPIO_IDR_0) == GPIO_IDR_0); //attende mentre USER è premuto
            cont++;
            GPIOE->ODR = (cont<<8);                            //cont traslato a sinistra
        }
    }
}
```

Contatore binario a 8 bit

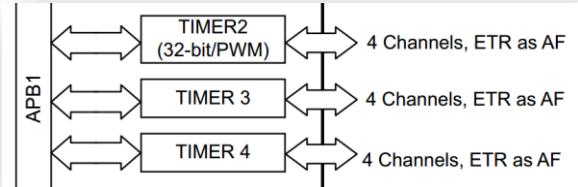
# 5. Timer

## 5.1 Timer general-purpose

I timer general-purpose TIM2, TIM3 e TIM4 sono contatori rispettivamente a 32, 16 e 16 bit.

L'unità time-base include:

- **TIMx\_CNT** (*Counter Register*)
- **TIMx\_PSC** (*Prescaler Register*)
- **TIMx\_ARR** (*Auto-Reload Register*)



Il **CNT** contiene il valore del conteggio, che comincia a contare quando viene abilitato.

L'**ARR** è precaricato con un certo valore, che rappresenta il numero di conteggi da effettuare prima di azzerare il contatore. Quando si modifica l'ARR, il suo contenuto può venir letto all'evento di update (alla fine del ciclo) o immediatamente, a seconda se il registro è bufferizzato o meno.

Il **PSC** serve per dividere la frequenza di clock, qualora fosse troppo alta.

$$T_{PSC} = (PSC + 1) \cdot T_{ck}$$

Ese.:  $T_{ck} = 1 \mu s$   
 $\Delta t = 1 s$

Il numero di conteggi è  $n = \Delta t / T_{ck} = 1.000.000$

Dunque ARR dovrebbe contenere il valore 1 milione.

- Se il timer è a 32 bit, conta fino a circa 4 miliardi => tutto OK.
- Se il timer è a 16 bit, conta fino a 65535, e 1 milione non c'entra.

Entra in gioco il PSC, che cambia il periodo di clock, per esempio, in:  $T_{PSC} = 1000$   $T_{ck} = 1 ms$

Il numero di conteggi è  $n = \Delta t / T_{PSC} = 1.000$

Il timer a 16 bit riesce a contare fino a mille => tutto OK.  
(è però peggiorata la risoluzione)

Ci sono 2 modalità in cui opera il timer:

- **base dei tempi**: aspetta intervalli di tempo uguali prima di una determinata operazione.  $\Delta t$  è prefissato.
- **contatore**: può contare in avanti, all'indietro, o alternando avanti e indietro. È usato per misurare il tempo.  $\Delta t$  è un'incognita.

$$\Delta t =_q N_{cont} T_{ck}$$

**Nota:** per i segnali PWM (simili ad onda quadra ma con durata del tetto e periodo variabili) occorrono 2 timer: uno per misurare la durata del tetto e uno per il periodo.

## 5.2 Registri utili dei timer

Oltre a CNT, PSC e ARR sono utili i seguenti registri:

- **TIMx\_CR1 (Control Register 1)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	UIF RE-MAP	Res.	CKD[1:0]	ARPE	CMS	DIR	OPM	URS	UDIS	CEN		
				rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- ARPE (*Auto-Reload Preload Enable*)
  - 0: ARR non bufferizzato (è letto immediatamente)
  - 1: ARR bufferizzato (è letto all'evento di update)
- CEN (*Counter Enable*)
  - 1: conteggio abilitato (comincia a contare)

- **TIMx\_CR2 (Control Register 2)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TI1S	MMS[2:0]	CCDS	Res.	Res.	Res.									
								rw	rw	rw	rw	rw			

- MMS (*Master Mode Selection*)
  - 001: Enable – genera un trigger TRGO quando CEN = 1
  - 010: Update – genera un trigger TRGO ad ogni evento di update

- **TIMx\_SR (Status Register)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CC4OF	CC3OF	CC2OF	CC1OF	Res.	Res.	TIF	Res.	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

- UIF (*Update Interrupt Flag*)
 

È settato dall'hardware quando si verifica un update, rimane 1 finché non è settato a 0 da software

  - 1: si è verificato un update (es. overflow: il conteggio è ricominciato)

- **TIMx\_DIER (DMA/Interrupt Enable Register)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res.	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res.	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

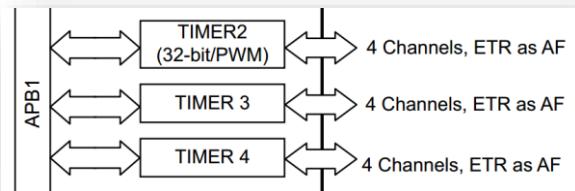
- UIE (*Update Interrupt Enable*)
  - 0: interruzioni disabilitate
  - 1: interruzioni abilitate

## 5.3 Progetto 6: LED lampeggianti

Vogliamo che i LED lampeggino, alternando mezzo secondo di accensione e mezzo secondo di spegnimento. Sfrutteremo quindi un timer in modalità base dei tempi, poiché abbiamo un  $\Delta t$  prefissato di 0,5 secondi al cui termine si compie una determinata operazione.

Per sicurezza sceglieremo il timer **TIM2**, così siamo certi che, essendo a 32 bit, riesca a raggiungere il conteggio.

Così come si fa per abilitare le GPIO, bisogna abilitare il timer. Dobbiamo dunque prendere il registro RCC che controlla il bus APB1 e attivare TIM2.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	DAC EN	PWR EN	Res	Res	CAN EN	Res	USB EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	Res
		rw	rw			rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Res	Res	WWD GEN	Res	Res	Res	Res	Res	TIM7E N	TIM6EN	Res	TIM4EN	TIM3EN	TIM2 EN
rw	rw			rw						rw	rw		rw	rw	rw

La frequenza di clock fornita dalla scheda è di 8 MHz, cioè 8 milioni di conteggi al secondo. Poiché vogliamo che i LED cambino di stato ogni mezzo secondo, il contatore deve arrivare a 4 milioni.

$$f_{ck} = 8 \text{ MHz}$$

$$T_{ck} = 1/f_{ck} = 125 \text{ ns}$$

$$N = \Delta t / T_{ck} = 4.000.000$$

→ Il valore di **TIM2\_ARR** è 4.000.000

Ovviamente per un valore così grande conviene usare il timer **TIM2**, che ha 32 bit a disposizione. Se volessimo usare i timer a 16 bit bisogna necessariamente mettere in gioco anche il prescaler (PSC).

TIM2 arriva infatti a circa 500 secondi, mentre TIM3 e TIM4 arrivano intorno a 8 millisecondi.

Per far partire effettivamente il timer, bisogna abilitare il conteggio e, per sicurezza, azzerarlo.

Dunque alziamo il bit CEN (*Counter Enable*) del registro CR1 e mettiamo 0 nel registro CNT.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	UIF RE-MAP	Res.	CKD[1:0]		ARPE	CMS	DIR	OPM	URS	UDIS	CEN	
				rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

A questo punto siamo pronti a ragionare sull'algoritmo che permette ai LED di "blinkare".

- ➔ SE il timer ha raggiunto il conteggio e i led sono *spenti*, ALLORA *accendi* i led e fai ripartire il conteggio
- ➔ SE il timer ha raggiunto il conteggio e i led sono *accesi*, ALLORA *spegni* i led e fai ripartire il conteggio

Ci sono molteplici modi per implementare questo algoritmo, ad esempio possiamo controllare che il contenuto di CNT sia maggiore del contenuto di ARR, oppure possiamo controllare che il registro di stato **SR** abbia rivelato un update. Procediamo, ad esempio, col secondo metodo.

Se CNT arriva a 4 milioni, che è il contenuto di ARR, è generato un evento di update. Questo evento alza il bit UIF del registro di stato, che rimane alto finché non lo riabbassiamo noi.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CC4OF	CC3OF	CC2OF	CC1OF	Res.	Res.	TIF	Res.	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Quindi se l'ultimo bit di SR è 1, allora accende o spegne i LED, a seconda se siano spenti o accesi.

Per decidere quale delle due cose fare, possiamo ricorrere a un **flag**, che vale:

- 0: LED spenti
- 1: LED accesi

In questo modo, quando flag=0 deve accendere, quando flag=1 deve spegnere. Non dimentichiamo poi di pulire l'UIF, cambiare il flag e far ripartire il conteggio da 0.

Quindi, se UIF=1 e i led sono spenti (accesi):

- Accendi (spegni) i led
- Pulisci UIF
- Aggiorna il flag

```

#include <stm32f30x.h>

short int flag = 0; //per segnalare accensione/spegnimento dei LED

void main(){
    RCC->AHBENR |= RCC_AHBENR_GPIOEEN; //per la porta E
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; //per il tim2
    GPIOE->MODER = 0x55550000; //output tutti i LED

    TIM2->CR1 |= TIM_CR1_CEN; //abilita il conteggio
    TIM2->ARR = 4000000; //mezzo secondo
    TIM2->CNT = 0; //azzera il conteggio

    while(1){
        if((TIM2->SR & TIM_SR UIF)==TIM_SR UIF && !flag){ //se ha raggiunto il conteggio ed è spento
            GPIOE->ODR = 0x0000FF00; //accende i LED
            TIM2->SR &= ~TIM_SR UIF; //azzera l'UIF (Update Interrupt Flag)
            flag = 1; //segnala che è acceso
        }
        if((TIM2->SR & TIM_SR UIF )==TIM_SR UIF && flag){ //se ha raggiunto il conteggio ed è acceso
            GPIOE->ODR = 0; //spegne i LED
            TIM2->SR &= ~TIM_SR UIF; //azzera l'UIF (Update Interrupt Flag)
            flag = 0; //segnala che è spento
        }
    }
}

```

*LED lampeggianti*

## 5.4 Progetto 7: misurazione intervallo di tempo

Vogliamo misurare la durata dell'intervallo di tempo in cui premiamo il pulsante USER.

Stesso discorso di prima: poiché con la frequenza di clock predefinita TIM2 conta fino a 500 secondi circa e gli altri fino ad 8 millisecondi circa, usiamo il TIM2 per evitare di lavorare anche con il prescaler.

L'algoritmo è piuttosto semplice:

- SE il pulsante è premuto, FINCHÉ è premuto conta, APPENA è stato rilasciato blocca il contatore e segna il valore del conteggio

Chiamiamo  $\tau$  la durata dell'intervallo  $\Delta t$  che dobbiamo misurare

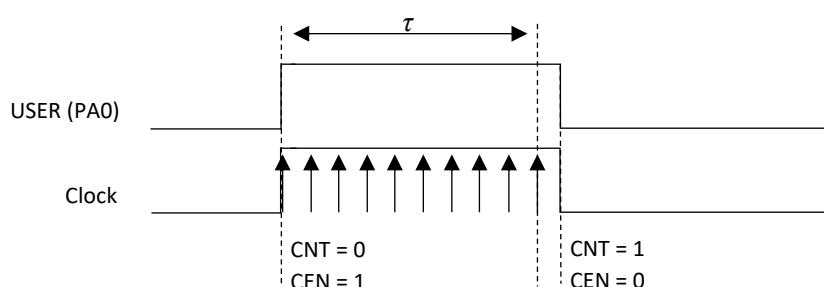
$$\tau =_q N_{\text{cont}} T_{\text{ck}}$$

dove N è il numero di conteggi che effettua in  $\Delta t$ .

Ricordiamo che  $T_{\text{ck}} = 125 \text{ ns}$ , quindi una volta saputo il numero di conteggi  $N_{\text{cont}}$  basta moltiplicarlo per  $125 \cdot 10^{-9}$

Quindi, se USER è premuto:

- Abilita il conteggio
- Azzera il conteggio
- Attende (conta)
- Disabilita il conteggio
- Calcola  $\tau$

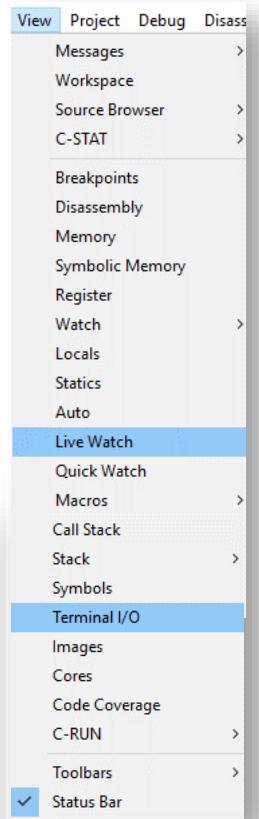
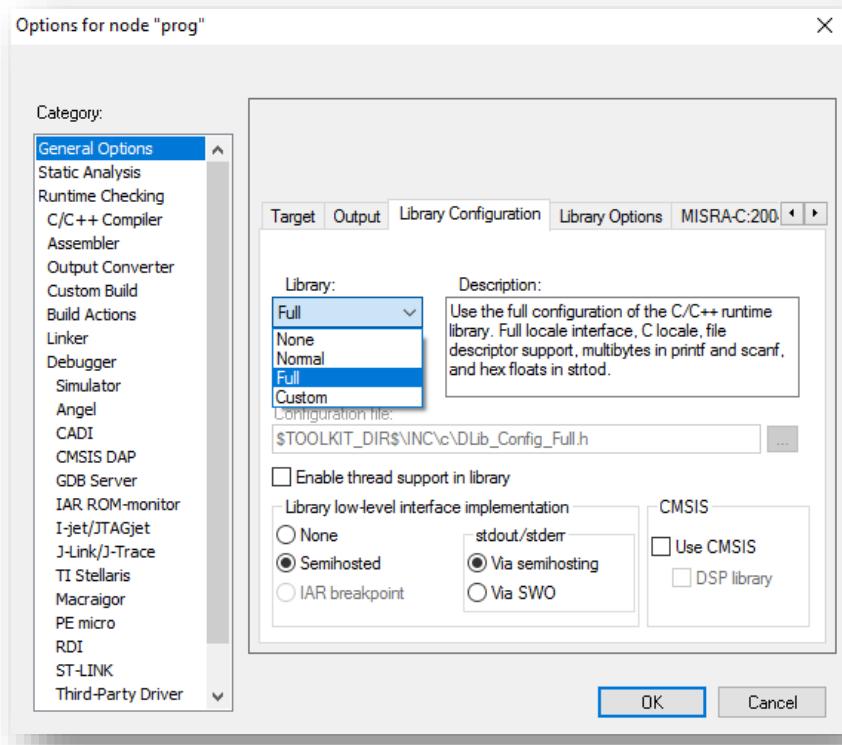


**Nota:** Ci sono 2 modi per visualizzare in tempo reale  $\tau$ :

- Attraverso la *Live Watch*: durante il debugging “View > Live Watch” e aggiungere la variabile tau (che deve essere globale per questa modalità)
- Attraverso il *Terminale I/O*: durante il debugging “View > Terminal I/O” (per stamparlo a video con printf)

Per poter usare la funzione *printf*, occorre includere la libreria **<stdio.h>** e andare nelle opzioni di progetto:

- General Options > Library Configuration > Library > Full



```

#include <stdio.h>
#include <math.h>
#include <stm32f30x.h>

#define Tck 125*pow(10,-9)           //Tck = 125 ns

float tau = 0;                  //è globale per la Live Watch

void main(){
    int N_cont;
    int count=1;

    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;           //per la porta A
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;          //per il tim2

    while(1){
        if((GPIOA->IDR & GPIO_IDR_0) == GPIO_IDR_0){ //se USER è premuto

            TIM2->CR1 |= TIM_CR1_CEN;                //abilita il conteggio
            TIM2->CNT = 0;                          //azzerà il conteggio

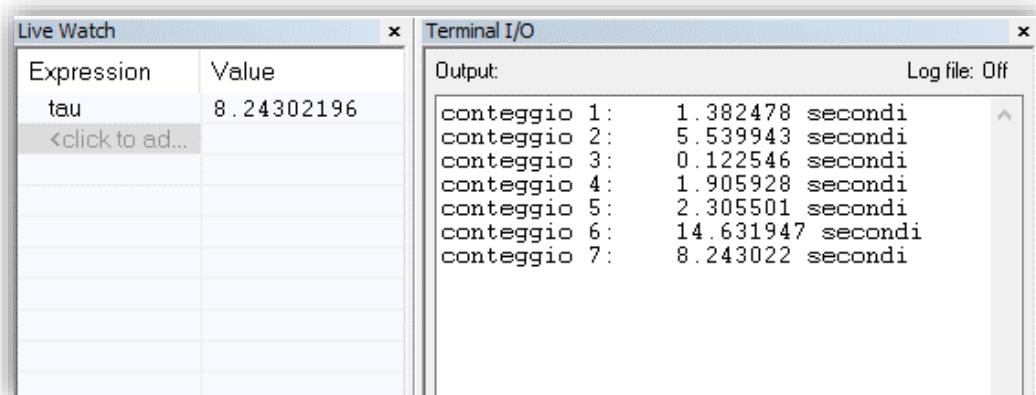
            while((GPIOA->IDR & GPIO_IDR_0) == GPIO_IDR_0); //attende mentre USER è premuto

            TIM2->CR1 &= ~TIM_CR1_CEN;              //disabilita il conteggio
            N_cont = TIM2->CNT;
            tau = (float)N_cont*Tck;

            printf("conteggio %d:\t%f secondi\n", count, tau);
            count++;
        }
    }
}

```

Misurazione intervallo di tempo



# 6. Interruzioni (NVIC)

## 6.1 Controllore delle interruzioni (NVIC)

Sappiamo già che quando avviene un'interruzione, viene eseguita la sua ISR (Interrupt Service Routine), cioè il codice che stabilisce cosa deve fare l'interrupt. L'**NVIC** (*Nested Vectored Interrupt Controller*) fa in modo che a seconda della causa dell'interruzione si esegua un'istruzione diversa. In altre parole, salta ad un indirizzo specifico a seconda della causa dell'interrupt.

L'NVIC gestisce 82 diverse interruzioni esterne, ciascuna delle quali è "attivabile" attraverso i registri **NVIC\_ISER [0] - [7]**.

Questi registri ISER (*Interrupt Set Enable Registers*) sono a 32 bit, perciò:

- ISER [0] per le interruzioni 0-31
- ISER [1] per le interruzioni 32-63
- ISER [2] per le interruzioni 64-81

Esiste una tabella (pag. 184 del Reference Manual) che associa un numero da 0 a 81 alle cause di interruzione.

Ad esempio l'interrupt generata dal timer TIM2 è la numero 28:

Table 6-1 NVIC registers				
Address	Name	Type	Reset	Description
0xE000E004	ICTR	RO	-	<i>Interrupt Controller Type Register; ICTR</i>
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	0x00000000	Interrupt Set-Enable Registers
0xE000E180 - 0xE000E19C	NVIC_ICER0 - NVIC_ICER7	RW	0x00000000	Interrupt Clear-Enable Registers
0xE000E200 - 0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	RW	0x00000000	Interrupt Set-Pending Registers
0xE000E280 - 0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	RW	0x00000000	Interrupt Clear-Pending Registers
0xE000E300 - 0xE000E31C	NVIC_IABR0 - NVIC_IABR7	RO	0x00000000	Interrupt Active Bit Register
0xE000E400 - 0xE000E41F	NVIC_IPR0 - NVIC_IPR59	RW	0x00000000	Interrupt Priority Register

Position	Priority	Type of priority	Acronym	Description	Address
21	28	settable	CAN_RX1	CAN_RX1 interrupt	0x0000 0094
22	29	settable	CAN_SCE	CAN_SCE interrupt	0x0000 0098
23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
24	31	settable	TIM1_BRK/TIM15	TIM1 Break/TIM15 global interrupts	0x0000 00A0
25	32	settable	TIM1_UP/TIM16	TIM1 Update/TIM16 global interrupts	0x0000 00A4
26	33	settable	TIM1_TRG_COM/TI M17	TIM1 trigger and commutation/TIM17 interrupts	0x0000 00A8
27	34	settable	TIM1_CC	TIM1 capture compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8
31	38	settable	I2C1_EV_EXTI23	I2C1 event interrupt & EXTI Line23 interrupt	0x0000 00BC

## 6.2 File startup\_stm32f303xc.s

Per poter usufruire del controllo delle interruzioni occorre aggiungere al progetto il file **startup\_stm32f303xc.s**.

Questo file è scritto in Assembly (.s) così da poter avere un pieno controllo sulla sua distribuzione in memoria, poiché con questo linguaggio le istruzioni vengono collocate in memoria una dopo l'altra così come vengono scritte.

Nel codice sono elencate le 82 interruzioni esterne, ecco un estratto:

```
    DCD    ADC1_2_IRQHandler           ; ADC1 and ADC2
    DCD    USB_HP_CAN1_RX_IRQHandler ; USB Device High Priority or CAN1 TX
    DCD    USB_LP_CAN1_RX0_IRQHandler; USB Device Low Priority or CAN1 RX0
    DCD    CAN1_RX1_IRQHandler       ; CAN1 RX1
    DCD    CAN1_SCE_IRQHandler      ; CAN1 SCE
    DCD    EXTI9_5_IRQHandler       ; External Line[9:5]
    DCD    TIM1_BRK_TIM15_IRQHandler; TIM1 Break and TIM15
    DCD    TIM1_UP_TIM16_IRQHandler ; TIM1 Update and TIM16
    DCD    TIM1_TRG_COM_TIM17_IRQHandler; TIM1 Trigger and Commutation and TIM17
    DCD    TIM1_CC_IRQHandler        ; TIM1 Capture Compare
    DCD    TIM2_IRQHandler          ; TIM2
    DCD    TIM3_IRQHandler          ; TIM3
    DCD    TIM4_IRQHandler          ; TIM4
    DCD    I2C1_EV_IRQHandler       ; I2C1 Event
```

```
PUBWEAK TIM1_CC_IRQHandler
SECTION .text:CODE:REORDER:NOROOT(1)
TIM1_CC_IRQHandler
    B TIM1_CC_IRQHandler

PUBWEAK TIM2_IRQHandler
SECTION .text:CODE:REORDER:NOROOT(1)
Loop
TIM2_IRQHandler
    B TIM2_IRQHandler

PUBWEAK TIM3_IRQHandler
SECTION .text:CODE:REORDER:NOROOT(1)
TIM3_IRQHandler
    B TIM3_IRQHandler

PUBWEAK TIM4_IRQHandler
SECTION .text:CODE:REORDER:NOROOT(1)
TIM4_IRQHandler
    B TIM4_IRQHandler
```

Il termine **PUBWEAK** (pubblica debole) indica che non esegue il codice al di sotto se trova un altro sottoprogramma con lo stesso nome. In parole poche, questo codice ha una priorità più bassa, quindi se non esiste un'altra funzione con il nome **TIM2\_IRQHandler** continua le istruzioni sottostanti e va in loop.

Quando è rilevata un'interruzione, viene dunque eseguito il codice relativo a quel preciso tipo di interruzione.

**Nota:** il file non è aggiunto al progetto tramite la direttiva `#include`, ma cliccando col tasto destro sul progetto e poi su *Add > Add Files*

## 6.3 File system\_stm32f30x.c

Aggiungendo solo lo startup\_stm32f303xc.s, tuttavia, si otterrà un errore di file mancante. Per risolvere il problema si deve aggiungere al progetto anche il file **system\_stm32f30x.c**.

Questo file permette anche di sfruttare la potenza della scheda al meglio, andando ad aumentare la frequenza di clock da 8 MHz a 72 MHz, cioè 72 milioni di conteggi al secondo.

Come fa a farlo? Vediamo nel dettaglio un altro registro RCC: il registro **RCC\_CFGR** (*Clock Configuration Register*).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Res	Res	Res	MCOF	Res	MCO[2:0]			I2SSRC	USBPRE ES	PLLMUL[3:0]				PLL XTPRE	PLL SRC	
			r		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Res	Res	PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]			SWS[1:0]		SW[1:0]			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	r	r	rw	rw		

I 4 bit denominati **PLLMUL** (*Phase-Locked Loop Multiplication Factor*) contengono il valore che andrà a moltiplicare la frequenza di clock.

In particolare, in questa libreria, il PLLMUL vale 9, per questo si arriva agli  $8 \times 9 = 72$  MHz come detto sopra.

$$f_{ck} \rightarrow f_{ck} * 9$$

\* 5. This file configures the system clock as follows:

```
=====
*-----*
*      System Clock source          | PLL (HSE)
*-----*
*      SYSCLK(Hz)                  | 72000000
*-----*
*      HCLK(Hz)                   | 72000000
*-----*
*      AHB Prescaler              | 1
*-----*
*      APB2 Prescaler             | 1
*-----*
*      APB1 Prescaler             | 2
*-----*
*      HSE Frequency(Hz)          | 8000000
*-----*
*-----*          PLLMUL           | 9          *-----*
*-----*          PREDIV           | 1          *-----*
*-----*          HPRE             | 1          *-----*
*-----*
```

```
/* PLL configuration */
RCC->CFGR &= (uint32_t)((RCC_CFGR_PLLSRC | RCC_CFGR_PLLXTPRE | RCC_CFGR_PLLMULL));
RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_PREDIV1 | RCC_CFGR_PLLXTPRE_PREDIV1 | RCC_CFGR_PLLMULL9);
```

**Nota:** il file non è aggiunto al progetto tramite la direttiva `#include`, ma cliccando col tasto destro sul progetto e poi su *Add > Add Files*

## 6.4 Progetto 8: LED lampeggianti con uso delle interruzioni

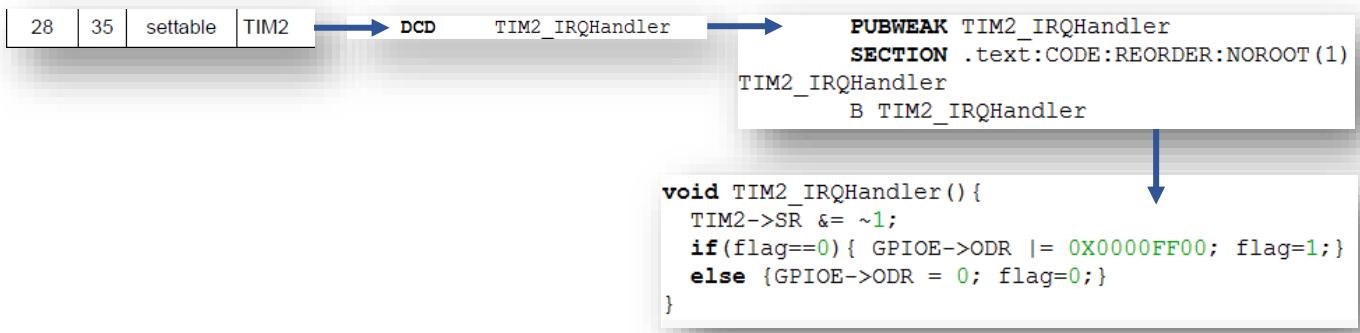
Vogliamo ancora che i LED lampeggino, ma questa volta sfruttando il meccanismo delle interruzioni.

L'esercizio si può fare in tanti modi e tante varianti, nel nostro caso realizzeremo un programma che fa accendere e spegnere i LED ogni mezzo secondo (ma si potrebbe anche, in maniera leggermente diversa, implementare un algoritmo che ogni mezzo secondo accenda i LED ma li tiene accesi solo per un periodo molto piccolo di tempo, magari gestendo l'attesa semplicemente con un ciclo for).

La prima differenza rispetto alla versione senza interruzioni è che bisogna abilitare le interruzioni del timer TIM2 alzando il bit UIE (*Update Interrupt Enable*) del registro DIER.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res.	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res.	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

Abbiamo già visto che la ISR relativa al timer è la numero 28, quindi bisogna alzare il 28° bit del NVIC\_ISER [0], così che venga eseguito il 28° sottoprogramma. Dobbiamo a questo punto realizzare l'effettiva ISR, e lo facciamo mediante una funzione che ha lo stesso nome dell'interruzione: **TIM2\_IRQHandler**.



Nel nostro caso, quando raggiunge mezzo secondo scatta UIF, e quindi UIE, e viene eseguita la nostra funzione handler, che non fa altro che abbassare l'UIF e accendere/spegnere i LED. Sfruttiamo un flag, che deve essere dichiarato globalmente per essere visibile dall'handler.

L'ultima cosa che rimane da fare è aggiornare l'ARR, poiché ti ricordo che nel system\_stm32f30x.c è stata modificata la frequenza di clock. Quindi se prima in mezzo secondo c'erano 4 milioni di conteggi, ora ce ne sono ben 4x9=36 milioni.

→ Il valore di **TIM2\_ARR** è **36.000.000**

```

#include <stm32f30x.h>

short int flag=0;

void main(){

    RCC->AHBENR |= RCC_AHBENR_GPIOEEN;      //porta E
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;      //timer
    GPIOE->MODER |= 0x55550000;                //abilita i LEDs

    TIM2->DIER |= TIM_DIER_UIE;              //abilita le interrupt nel timer
    NVIC->ISER[0] |= (1<<28);               //specifica la ISR

    TIM2->ARR = 36000000;                    //mezzo secondo
    TIM2->CNT = 0;                          //abilita il contatore

    while(1);
}

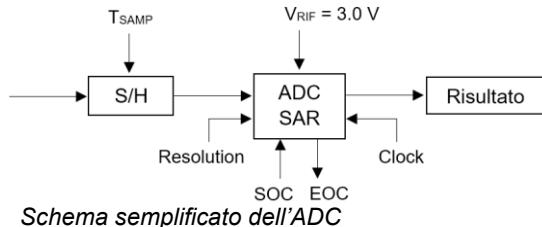
//--- ISR ---
void TIM2_IRQHandler()
{
    TIM2->SR &= ~TIM_SR UIF;                //resetta UIF
    if(flag==0){
        GPIOE->ODR |= 0x0000FF00;           //accende
        flag=1;
    }
    else {
        GPIOE->ODR = 0;                   //spegne
        flag=0;
    }
}

```

*LED lampeggianti*

# 7. Convertitore A/D

## 7.1 Descrizione generale



**SAR** = registro ad approssimazioni successive  
**SOC** = start of conversion  
**EOC** = end of conversion  
**S/H** = Sample & Hold

Schema semplificato dell'ADC

Se pensi che fino ad ora sia tutto semplice, è perché non hai ancora incontrato un ADC. Una sua configurazione errata può portare a errori difficili da individuare, quindi all'esame l'ADC può essere uno dei componenti che fa più danni, ma tutto sommato è un ostacolo facile da abbattere (un po' come succede in League of Legends).

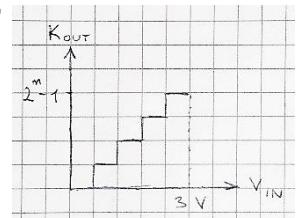
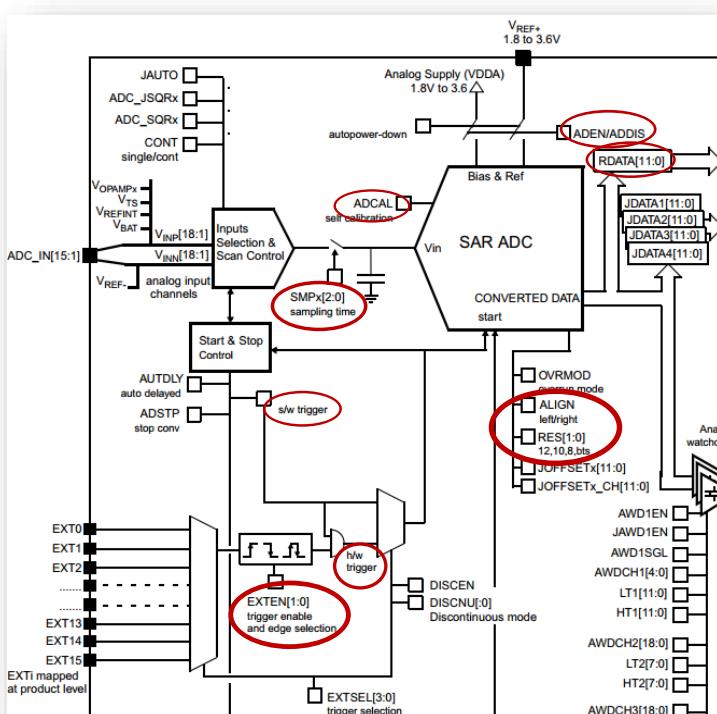
Un **ADC** riceve in ingresso una tensione e produce in uscita un codice.

La scheda è dotata di 4 ADC accoppiati (ADC1 e 2, ADC3 e 4) e mappati sul bus AHB. Possiamo scegliere come risoluzione tra 6-8-10-12 bit.

La conversione può avvenire in modalità singola o continua:

- In modalità singola (CONT=0) le conversioni vengono effettuate una volta sola.
- In modalità continua (CONT=1) le conversioni vengono effettuate ciclicamente.

Il contenuto di **RDATA** (*Regular Data*) è il numero di campioni contati, ognuno dei quali vale  $V_{DD}/2^n$  Volt, quindi la tensione (digitale) vera e propria è  $V_{IN} = RDATA * V_{DD}/2^n$ , dove  $V_{DD} = 3 \text{ V}$ .



**SMP** permette di regolare il tempo di sample

**RES** permette di configurare la risoluzione

**ALIGN** permette di scegliere dove allineare i bit nel registro da 16

**RDATA** contiene il risultato

**SW trigger** è per il SOC via SW

**HW trigger** è per il SOC via HW, posso configuralo con i 2 bit **EXTEN**

**ADCAL** permette la calibrazione dell'ADC

**ADEN - ADDIS** permettono l'abilitazione/disabilitazione dell'ADC

## Calibrazione e accensione ADC:

I canali possono essere configurati come *single-ended input* o *differential input*.

- *Single-ended input*: la tensione analogica da convertire è  $\text{ADC\_IN}_i - V_{\text{REF}}$ .
- *Differential input*: la tensione analogica da convertire è  $\text{ADC\_IN}_i - \text{ADC\_IN}_{i+1}$

La calibrazione precede qualsiasi operazione dell'ADC (per cui l'ADC deve essere disabilitato:  $\text{ADEN} = 0$ ). L'ADC calcola un fattore di calibrazione, che differisce a seconda se si opera in modalità single-ended o differenziale. Perciò si distingue:

- $\text{ADCALDIF} = 0$  per la calibrazione in modalità single-ended (default)
- $\text{ADCALFIF} = 1$  per la calibrazione in modalità differenziale

Prima di iniziare le operazioni dell'ADC bisogna:

- Abilitare la periferica ADC12 (o ADC34) dal `RCC_AHBENR`
- Abilitare il regolatore interno di tensione ( $\text{ADVREGEN} = 10 \rightarrow 00 \rightarrow 01$ )
- Attendere il tempo di startup ( $T_{\text{ADCVREG\_STUP}} = 10 \mu\text{s}$  al massimo)
- Scegliere come impostare `ADCALDIF` (default: *single-ended*)
- Avviare la procedura di calibrazione ( $\text{ADCAL} = 1$ )
- Aspettare il termine della calibrazione ( $\text{ADCAL} = 0$ )

Per effettuare la conversione:

- Abilitare l'ADC ( $\text{ADEN} = 1$ )
- Aspettare che sia pronto ( $\text{ADRDY} = 1$ , si può sfruttare l'interruzione, se abilitata con  $\text{ADRDYIE} = 1$ )
- Avviare la conversione ( $\text{ADSTART} = 1$ , aspetta il trigger esterno se  $\text{EXTEN} \neq 0$ )
- Aspettare il termine della conversione ( $\text{ADSTART} = 0$ )
- Disabilitare l'ADC ( $\text{ADDIS} = 1$ , questo abbassa  $\text{ADEN}$ )

Per interrompere precocemente le conversioni basta settare  $\text{ADSTP} = 1$

## Clock:

Come clock può essere scelto in due modi:

- Sfruttare il clock di coppia (`ADC12_CK` o `ADC34_CK`).  
Può arrivare fino a 72 MHz cambiando il valore di `PLLMUL` nel registro `RCC_CNFG`.  
Per questa soluzione bisogna impostare `CKMODE = 00`
- Sfruttare il clock del bus AHB.  
Può essere diviso per 1, 2, 4.  
Per questa soluzione bisogna impostare `CKMODE = 01, 10, 11` (diviso per 1, 2, 4)

## **Selezione canale:**

Ogni ADC è dotato di 18 canali. Di particolare interesse sono i seguenti canali:

- ADC1\_IN16 = sensore di temperatura
- ADC1\_IN17 = batteria esterna
- ADCx\_IN18 =  $V_{RIF}$

Si dice *gruppo* una sequenza di conversioni, il cui numero massimo è 16. L'ordine dei canali nella sequenza e il numero di conversioni vanno specificati nei registri ADC\_SQRx.

Alla fine di una conversione EOC = 1 e il risultato è memorizzato in ADC\_DR, mentre alla fine di tutta la sequenza EOS = 1.

EOC si abbassa quando il ADC\_DR viene letto.

## **Tempo di campionamento:**

Prima di iniziare una conversione, l'ADC aspetta un certo *tempo di campionamento (sampling time)* per permettere al condensatore di caricarsi al livello di tensione di ingresso.

Ovviamente più è alto questo tempo più è preciso il campionamento, ai danni della velocità di campionamento.

Esistono 8 possibili tempi  $T_{SAMP}$  (da 1.5 a 601.5 colpi di clock). Il tempo di campionamento va scritto nei bit SMP dei registri ADCx\_SMPR1 e 2.

Il tempo totale di conversione in (colpi di clock) è:

$$T_{CONV} = T_{SAMP} + T_{SAR}$$

$T_{SAR}$  dipende dalla risoluzione (12, 10, 8, 6 bit  $\rightarrow$  12.5, 10.5, 8.5, 6.5 colpi di clock).

Quando il campionamento è terminato l'ADC alza il bit EOSMP.

## **Trigger esterno:**

Per avviare la conversione tramite trigger esterno, non basta avere ADSTART=1.

Innanzitutto, il trigger deve essere abilitato tramite i bit EXTEN quando ancora ADSTART=0, successivamente bisogna alzare il bit ADSTART.

Dunque:

- Se EXTEN = 0, la conversione parte quando ADSTART = 1
- Se EXTEN ≠ 0, parte quando ADSTART = 1 e l'evento del trigger è verificato

Si possono scegliere fino a 16 eventi trigger e selezionarli tramite i bit EXTSEL (tabelle pag. 221 del Reference Manual).

## Sensore di Temperatura:

È in grado di misurare la temperatura da -40°C a 125°C.

Per usare il sensore di temperatura si deve:

- Selezionare il canale 16
- Impostare il tempo di sampling a più di 2.2 µs
- Settare TSEN nel registro CCR

Il risultato è  $T = 25 + \frac{1.43-V}{0.0043}$  (V è la tensione calcolata a partire dal DR)

## 7.2 Registri utili dell'ADC

I principali registri sono:

- **ADCx\_ISR (Interrupt and Status Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	JQOVF	AWD3	AWD2	AWD1	JEOS	JEOC	OVR	EOS	EOC	EOSMP	ADRDY
					r_w1	r_w1	r_w1	r_w1	r_w1	r_w1	r_w1	r_w1	r_w1	r_w1	r_w1

- EOS (*End Of Sequence*)
  - 1: la sequenza di conversioni è terminata
- EOC (*End Of Conversion*)
  - 1: la conversione è terminata
- EOSMP (*End Of Sampling*)
  - 1: il campionamento è terminato
- ADRDY (*ADC Ready*)
  - 1: l'ADC è pronto per la conversione (si alza dopo che ADEN=1)

- **ADCx\_IER (Interrupt Enable Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	JQ OVFIE	AWD3 IE	AWD2 IE	AWD1 IE	JEOSIE	JEOCIE	OVRIE	EOSIE	EOCIE	EOSMP IE	ADRDY IE
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Per abilitare le interruzioni di EOS, EOC, EOSMP, ADRDY

- **ADCx\_CR (Control Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AD CAL	ADCA LDIF	ADVREGEN[1:0]		Res.	Res.	Res.	Res.	Res.	Res.						
rs	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	JAD STP	AD STP	JAD START	AD START	AD DIS	AD EN
										rs	rs	rs	rs	rs	rs

- **ADCAL (ADC Calibration)**
  - 0: calibrazione completata
  - 1: avvio calibrazione, calibrazione in corso
- **ADCALDIF (ADC Calibration Differential mode)**
  - 0: modalità single-ended
  - 1: modalità differenziale
- **ADVREGEN (ADC Voltage Regulator Enable)**
  - 00: stato intermedio per cui si passa quando si abilita o disabilita V<sub>REG</sub>
  - 01: V<sub>REG</sub> abilitato
  - 10: V<sub>REG</sub> disabilitato
- **ADSTP (ADC Stop)**
  - 1: interrompe la conversione
- **ADSTART (ADC Start)**
  - 1: avvio conversione / conversione in corso
- **ADDIS (ADC Disable)**
  - 1: disabilitazione ADC
- **ADEN (ADC Enable)**
  - 0: ADC disabilitato (in seguito a ADDIS=1)
  - 1: ADC abilitato

- **ADCx\_CFGR (ADC Configuration Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	AWD1CH[4:0]				JAUTO	JAWD1EN	AWD1EN	AWD1SGL	JQM	JDISCEN	DISCNUM[2:0]			DISCEN	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	AUTDLY	CONT	OVRMOD	EXTEN[1:0]	EXTSEL[3:0]			ALIGN	RES[1:0]		Res.	DMACFG	DMAEN		
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw		rw			rw

- **CONT (Continuous mode)**
  - 0: conversione singola
  - 1: conversione continua
- **EXTEN (External Trigger Enable)**
  - 00: trigger HW disabilitato
  - 01, 10, 11: trigger HW abilitato (fronte di salita, discesa, entrambi)
- **ALIGN (Data Alignment)**
  - 0: allineamento a destra
  - 1: allineamento a sinistra
- **RES (Data Resolution)**
  - 00, 01, 10, 11: risoluzione di 12, 10, 8, 6 bit
- **DMAEN (DMA Enable)**
  - 0: DMA disabilitato
  - 1: DMA abilitato

- **ADCx\_SMPR1 e 2 (ADC Sampling Time Registers)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			Res.	Res.	Res.
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	SMP18[2:0]	SMP17[2:0]			SMP16[2:0]			SMP15[2:1]			
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

000: 1.5 ADC clock cycles  
 001: 2.5 ADC clock cycles  
 010: 4.5 ADC clock cycles  
 011: 7.5 ADC clock cycles  
 100: 19.5 ADC clock cycles  
 101: 61.5 ADC clock cycles  
 110: 181.5 ADC clock cycles  
 111: 601.5 ADC clock cycles

- SMPx: tempo di campionamento dell'x-esimo canale

- **ADCx\_SQR1, 2, 3 e 4 (ADC Sequence Registers)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	SQ4[4:0]				Res.	SQ3[4:0]				Res.	SQ2[4]		
			rw	rw	rw	rw	rw		rw	rw	rw	rw		rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ2[3:0]				SQ1[4:0]				Res.	L[3:0]						
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	SQ9[4:0]				Res.	SQ8[4:0]				Res.	SQ7[4]		
			rw	rw	rw	rw	rw		rw	rw	rw	rw		rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ7[3:0]				SQ6[4:0]				Res.	SQ5[4:0]						
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	SQ14[4:0]				Res.	SQ13[4:0]				Res.	SQ12[4]		
			rw	rw	rw	rw	rw		rw	rw	rw	rw		rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ12[3:0]				SQ11[4:0]				Res.	SQ10[4:0]						
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.															
RDATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw

Il risultato vero e proprio è RDATA \* V<sub>DD</sub>/2<sup>n</sup>

- **ADCxy\_CCR (ADC Common Control Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	VBAT EN	TS EN	VREF EN	Res.	Res.	Res.	CKMODE[1:0]
									rw	rw	rw				rw rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MDMA[1:0]		DMA CFG	Res.	DELAY[3:0]				Res.	Res.	Res.	DUAL[4:0]				
rw	rw	rw		rw	rw	rw	rw				rw	rw	rw	rw	

- **CKMODE (Clock Mode)**

- 00: usa il clock di coppia ADC\_CK12 o ADC\_CK34
- 01, 10, 11: usa il clock del bus AHB diviso per 1, 2, 4

## 7.3 Progetto 9: conversione tensione del pulsante USER

Vogliamo realizzare un programma che converta la tensione di ingresso della linea PA0 (il pulsante USER) in digitale. Per fare ciò, occorre innanzitutto impostare la PA0 in modalità analogica, dopodiché si può passare alla conversione del valore analogico letto su questa linea in un valore digitale.

La PA0 è collegata al canale 1 dell'ADC1 (ADC1\_IN1), per cui il convertitore che dobbiamo utilizzare è l'ADC1 e in particolare la sequenza di conversione sarà costituita da un'unica conversione, quella del canale 1.

Pin number			Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
LQF P100	LQF P64	LQF P48					Alternate functions	Additional functions
23	14	10	PA0	I/O	TTa		USART2_CTS, TIM2_CH1_ETR, TIM8_BKIN <sup>(3)</sup> , TIM8_ETR <sup>(3)</sup> , TSC_G1_IO1, COMP1_OUT	ADC1_IN1, COMP1_INM, RTC_TAMP2, WKUP1, COMP7_INP <sup>(3)</sup>

Prima della conversione vera e propria bisogna fare delle operazioni preliminari di calibrazione, abilitazione e configurazione, che abbiamo già visto in maniera teorica.

Potremo aver bisogno anche di un timer per attendere i 10 µs di startup, ma è sufficiente usare al suo posto un nabbissimo ciclo for (per pigrizia).

La conversione la eseguiamo in modalità continua (CONT = 1), in modo tale che converte “infinite” volte il valore. ADSTART pertanto sarà settato solo una volta.

Infine, dobbiamo ricordarci che il contenuto di RDATA è il numero di campioni contati, ognuno dei quali vale  $V_{DD}/2^n$  Volt, quindi la tensione vera e propria è  $V_{DD}/2^n$  Volt per il numero di campioni.

Per semplicità, descrivo il programma schematicamente:

1. Abilitazione ADC12, GPIOA
2. PA0 in modalità analogica (MODER0 = 11)
3. Setup dell'ADC:
  - Abilitazione regolatore di tensione
    - i. Abilitare il regolatore (10 -> 00 -> 01)
    - ii. Attendere il tempo di startup (10 µs)
  - Configurazione clock (CKMODE = 01, clock del bus AHB)
  - Calibrazione ADC
    - i. Avviare la calibrazione (ADCAL = 1)
    - ii. Aspettare il termine della calibrazione (ADCAL = 0)
  - Abilitazione ADC
    - i. Abilitare l'ADC (ADEN = 1)
    - ii. Aspettare che sia pronto (ADRDY = 1)

```
// Abilitazione regolatore di tensione
ADC1->CR &= ~ADC_CR_ADVREGEN_1;
ADC1->CR |= ADC_CR_ADVREGEN_0;
for(int i=0; i<1000; i++);
```

```
// Configurazione clock
ADC1_2->CCR |= ADC12_CCR_CKMODE_0;
```

```
// Calibrazione ADC
ADC1->CR |= ADC_CR_ADCAL;
while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL);
```

```
// Abilitazione ADC
ADC1->CR |= ADC_CR_ADEN;
while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD);
```

- Configurazione ADC
  - i. Registro CFGR
  - ii. Registri SQR
  - iii. Registri SMP

```
// Configurazione ADC
ADC1->CFGGR |= ADC_CFGGR_CONT;
ADC1->SQR1 = ADC_SQR1_SQ1_0;
ADC1->SQR1 &= ~ADC_SQR1_L;
ADC1->SMPR1 |= ADC_SMPR1_SMP1;
```

```
//CONT=1, conversione continua
//SQ1=00001: canale 1 (PA0)
//L=0: 1 conv.
//SMP1=111, 601.5 CK
```

- Conversione:
  - Avviare conversione (ADSTART = 1)
  - Aspettare il termine della conversione (EOC = 1)
- Lettura risultato
- Disabilitazione ADC (ADDIS = 1)

**Nota:** In realtà non lo disabilitiamo perché nel nostro programma convertiamo “all’infinito”

**Nota:** Dovendo effettuare la conversione di un unico canale, per aspettare il termine della conversione si può sia controllare EOC che ADSTART

**Nota:** 3 / 4096 è una divisione di interi, per cui si ottiene 0. Bisogna quindi scrivere 3.0 / 4096.0

**Nota importante:** Il clock si deve impostare quando l’ADC è disabilitato, le altre configurazioni vanno effettuate quando l’ADC è abilitato

```
#include <stm32f30x.h>

float risultato;

void ADCsetup();

void main() {
    //1. Abilitazione ADC12, GPIOA
    RCC->AHBENR |= RCC_AHBENR_ADC12EN | RCC_AHBENR_GPIOAEN;

    //2. PA0 in modalità analogica (MODER0=11)
    GPIOA->MODER |= GPIO_MODE_MODER0;

    //3. Setup dell'ADC
    ADCsetup();

    //4. Conversione
    ADC1->CR |= ADC_CR_ADSTART; //ADSTART=1, avvio conversione

    while (1) {

        while ((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC); //attesa EOC=1

        //5. Lettura risultato
        risultato = ADC1->DR*(3.0/4096.0); //risultato * VDD/2^n
    }
}
```

Conversione tensione del pulsante USER

```

void ADCsetup()
{
    // Abilitazione regolatore di tensione
    ADC1->CR |= ~ADC_CR_ADVREGEN_1;                                //ADVREGEN = 10->00
    ADC1->CR |= ADC_CR_ADVREGEN_0;                                 //ADVREGEN = 00->01
    for(int i=0; i<1000; i++);                                     //attesa di 10 µs

    // Configurazione clock
    ADC1_2->CCR |= ADC12_CCR_CKMODE_0;                            //CKMODE=01, clock del bus AHB

    // Calibrazione ADC
    ADC1->CR |= ADC_CR_ADCAL;                                     //ADCAL=1, avvio calibrazione
    while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL);           //attesa ADCAL=0

    // Abilitazione ADC
    ADC1->CR |= ADC_CR_ADEN;                                     //ADEN=1, abilitazione ADC
    while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD);          //attesa ADRDY=1

    // Configurazione ADC
    ADC1->CFGGR |= ADC_CFGGR_CONT;                               //CONT=1, conversione continua
    ADC1->SQR1  = ADC_SQR1_SQ1_0;                                //SQ1=00001: canale 1 (PA0)
    ADC1->SQR1  |= ~ADC_SQR1_L;                                  //L=0: 1 conv.
    ADC1->SMPR1 |= ADC_SMPR1_SMP1;                             //SMP1=111, 601.5 CK
}

```

Live Watch	
Expression	Value
risultato	1.46484375E-3

Pulsante non premuto: tensione  
prossima a 0

Live Watch	
Expression	Value
risultato	2.9978027344

Pulsante premuto: tensione  
prossima a  $V_{DD} = 3 V$

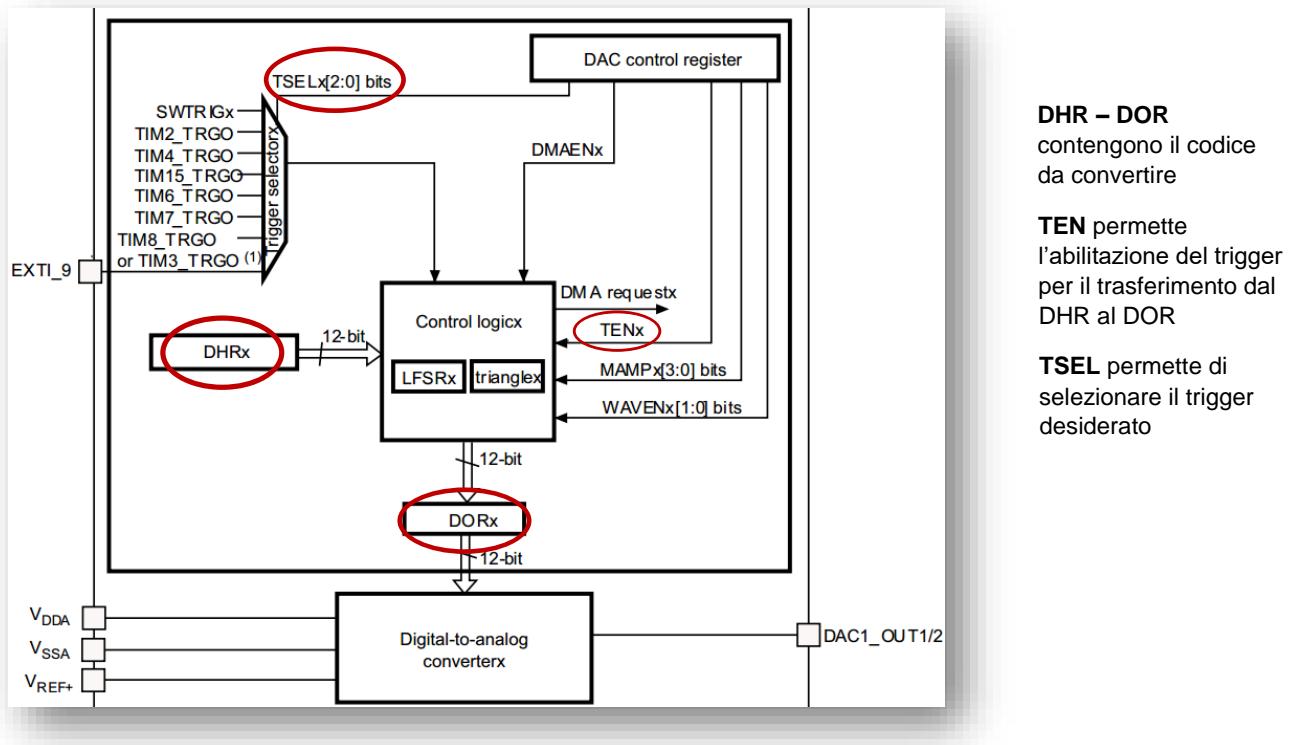
# 8. Convertitore D/A

## 8.1 Descrizione generale

Un **DAC** riceve in ingresso un codice e produce in uscita una tensione.

La scheda è dotata di un solo DAC (DAC1), che può essere usato a 8 o 12 bit e mappato sul bus APB1.

Due canali di uscita DAC1\_OUT1, DAC1\_OUT2 (PA4 e PA5), che possono essere utilizzati separatamente (single-mode) o insieme (dual-mode). Devono essere impostati come linee analogiche.



I 2 canali sono abilitati tramite i bit EN1 e EN2. In modalità single-mode (che è quella che useremo) il canale 2 non è disponibile.

Il DAC include anche un buffer di output per ognuno dei 2 canali, che può essere disabilitato con il bit BOFF. Il buffer è utile per mantenere la tensione generata.

Il codice input va scritto nei registri DHR8Rx, DHR12Lx, DHR12Rx, a seconda del numero di bit (8 o 12) e dell'allineamento del dato (destra o sinistra). Il dato scritto è poi caricato nel vero e proprio registro **DHR** (*Data Holding Register*), che non si trova sul Reference Manual perché è interno e non è mappato in memoria.

Il codice è trasferito poi dal DHR al **DOR** (*Data Output Register*), su cui non abbiamo il permesso di scrivere direttamente. Questo trasferimento può avvenire o immediatamente (dopo 1 colpo di

clock), o tramite trigger software (alzando il bit SWTRIG) o tramite trigger hardware (dopo 3 colpi di clock).

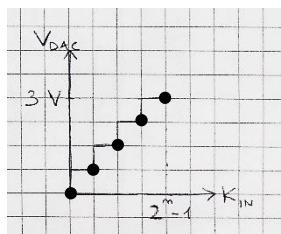
Per abilitare il trigger software o hardware va settato il bit TEN (*Trigger Enable*).

L'evento del trigger può dipendere da varie sorgenti, che si possono selezionare tramite i bit TSEL (*Trigger Selection*).

Una volta caricato il DOR, ha inizio la generazione della tensione. Il risultato è disponibile solo dopo un certo tempo  $t_{SETTLING}$ .

La tensione in uscita è determinata dal DAC secondo l'equazione:

$$V_{OUT} = V_{DD} * DOR / 2^n - 1$$



## 8.2 Registri utili del DAC

I principali registri sono:

- **DAC\_CR (Control Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	DMAU DRIE2	DMA EN2	MAMP2[3:0]		WAVE2[1:0]		TSEL2[2:0]		TEN2	BOFF2	EN2			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Res.	Res.	DMAU DRIE1	DMA EN1	MAMP1[3:0]		WAVE1[1:0]		TSEL1[2:0]		TEN1	BOFF1	EN1			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- DMAENx (*DMA Enable*)
  - 0: DMA disabilitato
  - 1: DMA abilitato
- TSELx (*Trigger selection*)
  - I 4 trigger più interessanti:
    - 001: timer TIM3 (DAC\_TRIG\_RMP = 1 nel registro SYSCFG\_CFGR1, dopo aver abilitato il bit SYSCFGEN nel registro RCC\_APB2ENR)
    - 101: timer TIM4
    - 100: timer TIM2
    - 111: trigger software (SWTRIG = 1)
- TENx (*Trigger Enable*)
  - 0: trigger disabilitati
  - 1: trigger abilitati
- BOFFx (*Buffer Disable*)
  - 0: buffer abilitato
  - 1: buffer disabilitato

- **DAC\_SWTRIGR (Software Trigger Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.														
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	SWTRIG2	SWTRIG1													
														w	w

- SWTRIGx (*Software Trigger*)
  - 0: trigger software disabilitato
  - 1: trigger software abilitato

- **DAC\_DHR8R1** (*Data Holding Register – 8 bit Right Aligned*)  
**DAC\_DHR12R1** (*Data Holding Register – 12 bit Right Aligned*)  
**DAC\_DHR12L1** (*Data Holding Register – 12 bit Left Aligned*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DACC1DHR[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	DACC1DHR[11:0]								rw	rw	rw	rw
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DACC1DHR[11:0]	Res.	Res.	Res.	Res.	Res.										
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw			

- **DAC\_DOR1** (*Data Output Register*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	DACC1DOR[11:0]								r	r	r	r
				r	r	r	r	r	r	r	r	r	r	r	r

## 8.3 Progetto 10: DAC – ADC in serie

Un esercizio solo sul convertitore D/A non avrebbe senso per noi, perché poi per leggere il valore della tensione in output ci occorrerebbe un convertitore A/D. Quindi l'esercizio che faremo consisterà nella serie dei due convertitori: l'obiettivo è generare una tensione col DAC e acquisirla con l'ADC.

Supponendo di lavorare a 12 bit, dovremo dunque inserire in ingresso al DAC un codice (da 0 a 4095) e ricevere in uscita all'ADC lo stesso codice (in teoria, nella pratica la tensione non è precisamente fissa).

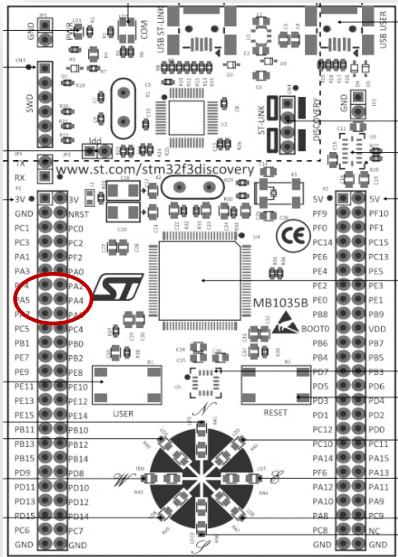
Per poter lavorare in questo modo, dobbiamo collegare fisicamente l'uscita del DAC all'ingresso dell'ADC.

Sappiamo che DAC1 ha 2 pin di uscita:

- PA4 per DAC1\_OUT1
- PA5 per DAC1\_OUT2

Stiamo operando in single-mode, quindi ci serve solo il canale 1, e quindi PA4.

PA4	I/O	TTa	SPI1_NSS, SPI3_NSS/I2S3_WS <sup>(3)</sup> , USART2_CK, TSC_G2_IO1, TIM3_CH2	ADC2_IN1(DAC1_OUT1) OPAMP4_VINP, COMP1_INM4, COMP2_INM4, COMP3_INM4, COMP4_INM4, COMP5_INM4, COMP6_INM4, COMP7_INM4
PA5	I/O	TTa	SPI1_SCK, TIM2_CH1_ETR, TSC_G2_IO2	ADC2_IN2(DAC1_OUT2) <sup>(3)</sup> , OPAMP1_VINP, OPAMP2_VINM, OPAMP3_VINP, COMP1_INM5, COMP2_INM5, COMP3_INM5, COMP4_INM5, COMP5_INM5, COMP6_INM5, COMP7_INM5



Vediamo sul nostro dispositivo dove si trova il pin della porta PA4.

Dovremo usare un jumper per collegarlo all'ingresso dell'ADC, perciò quest'ingresso deve essere adiacente a PA4.

Usiamo, ad esempio, PA2, che è collegato all'ingresso 3 di ADC1.



PA2	I/O	TTa	USART2_TX, TIM2_CH3, TIM15_CH1, TSC_G1_IO3, COMP2_OUT	ADC1_IN3(COMP2_INM, AOP1_OUT)
-----	-----	-----	---	----------------------------------

La parte relativa al DAC è molto semplice:

- Attivare il clock del DAC dal registro RCC\_APB1ENR
- DAC1\_OUT1 (PA4) e ADC1\_IN3 (PA2) in modalità analogica
- Abilitare eventualmente i trigger (TEN = 1)
- Scegliere l'eventuale trigger (TSEL)
- Abilitare il canale 1 (EN1 = 1)
- Scrivere il codice nel DHR

La parte relativa all'ADC è essenzialmente un copia e incolla dell'esercizio precedente con qualche piccola modifica come la selezione del canale 3 (PA2) al posto del canale 1 (PA0).

```

#include <stdio.h>
#include <stm32f30x.h>

// variabili del DAC
int codice_in = 500; //codice in ingresso [0; 4095]
float tensione_out; //stima della tensione generata dal DAC

// variabili dell'ADC
float tensione_in; //stima della tensione letta dall'ADC
int codice_out; //risultato dell'ADC

void abilitazione_periferiche();
void disabilitazione_periferiche();
void setup_ADC();
void setup_DAC();

void main(){
    // Abilitazione DAC1, ADC12, GPIOA
    abilitazione_periferiche();

    // PA4 e PA2 in modalità analogica
    GPIOA->MODER |= GPIO_MODER_MODER4;
    GPIOA->MODER |= GPIO_MODER_MODER2;

    //setup di DAC e ADC
    setup_DAC();
    setup_ADC();

    // Conversione
    ADC1->CR |= ADC_CR_ADSTART; //ADSTART=1, avvio conversione
    while((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC); //attesa EOC=1

    // Risultato (DAC)
    tensione_out = codice_in * (3.0/4095.0);

    printf("DAC\n");
    printf("ingresso: %d\n", codice_in);
    printf("uscita: %f V\n\n", tensione_out);

    // Risultato (ADC)
    codice_out = ADC1->DR;
    tensione_in = codice_out*(3.0/4096.0);

    printf("ADC\n");
    printf("ingresso: %f V\n", tensione_in);
    printf("uscita: %d\n", codice_out);

    // Disabilitazione ADC e DAC
    disabilitazione_periferiche();

    while(1);
}

```

DAC – ADC in serie

```

void abilitazione_periferiche(){
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //abilitazione GPIOA
    RCC->AHBENR |= RCC_AHBENR_ADC12EN; //abilitazione ADC12
    RCC->APB1ENR |= RCC_APB1ENR_DAC1EN; //abilitazione DAC1
}

void disabilitazione_periferiche(){
    ADC1->CR |= ADC_CR_ADDIS; //ADDIS=1
    DAC1->CR &= ~DAC_CR_EN1; //EN1=0
}

void setup_DAC(){
    DAC1->CR |= DAC_CR_EN1; //abilitazione canale 1
    DAC1->DHR12R1 = codice_in; // Scrittura codice nel DHR
    for(int i=0; i<1000; i++); //attesa generazione di tensione
}

void setup_ADC(){
    // Abilitazione regolatore di tensione
    ADC1->CR &= ~ADC_CR_ADVREGEN_1; //ADVREGEN = 10->00
    ADC1->CR |= ADC_CR_ADVREGEN_0; //ADVREGEN = 00->01
    for(int i=0; i<1000; i++); //attesa di 10 µs

    // Configurazione clock
    ADC1_2->CCR |= ADC12_CCR_CKMODE_0; //CKMODE=01, clock del bus AHB

    // Calibrazione ADC
    ADC1->CR |= ADC_CR_ADCAL; //ADCAL=1, avvio calibrazione
    while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL); //attesa ADCAL=0

    // Abilitazione ADC
    ADC1->CR |= ADC_CR_ADEN; //ADEN=1, abilitazione ADC
    while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD); //attesa ADRDY=1

    // Configurazione ADC
    ADC1->CFGGR &= ~ADC_CFGGR_CONT; //CONT=0, conversione singola
    ADC1->SQR1 = (3<<6); //SQ1=00011: canale 3 (PA2)
    ADC1->SQR1 &= ~ADC_SQR1_L; //L=0: 1 conv.
    ADC1->SMPR1 |= ADC_SMPR1_SMP3; //SMP3=111, 601.5 CK
}

```

Terminal I/O
Output:
DAC ingresso: 4095 uscita: 3.000000 V
ADC ingresso: 2.955322 V uscita: 4035

Terminal I/O
Output:
DAC ingresso: 2017 uscita: 1.477656 V
ADC ingresso: 1.478027 V uscita: 2018

Terminal I/O
Output:
DAC ingresso: 100 uscita: 0.073260 V
ADC ingresso: 0.073242 V uscita: 100

## 8.4 Progetto 11: generazione sinusoide

Vogliamo simulare una tensione alternata, generando una sinusoide con il DAC e acquisendola con l'ADC.

Per farlo, inseriamo manualmente i valori della sinusoide in un vettore  $V_{\text{SIN}}$ , che conterrà quindi i valori delle tensioni che il DAC deve generare per simulare un segnale sinusoidale.

Ovviamente l'acquisizione deve avvenire mentre il DAC genera la sinusoide, poiché non avrebbe senso farlo al termine della generazione, dato che a quel punto la tensione è costante e uguale all'ultimo valore ottenuto dal DAC.

Le conversioni del DAC e dell'ADC devono quindi avvenire nello stesso ciclo for (o while), perché se usassimo un primo ciclo per passare ogni valore del vettore al DAC e poi un secondo ciclo per l'acquisizione, l'ADC leggerebbe solo la tensione finale prodotta dal DAC, cosa che abbiamo appena detto di voler evitare.

Vogliamo inoltre che l'intervallo tra un campione e il successivo sia costante: il passaggio dal DHR al DOR deve essere regolato da un timer. Però verificare manualmente il flag UIF ad ogni ciclo del for potrebbe comunque generare i campioni in maniera irregolare, quindi per motivi "estetici" andrebbe usato il trigger del timer, che invia un segnale al DAC ad ogni evento di update.

In questo modo se anche impiegassimo 10 colpi di clock per controllare l'UIF, si sarebbero comunque generati 10 campioni.

Purtroppo, per dare un senso a quest'ultima cosa occorrerebbe il DMA, che permetterebbe al DAC di prendere "da solo" il prossimo valore della sinusoide ad ogni colpo di clock, senza passargli quindi i valori nel ciclo for.

In questa versione dell'esercizio non usiamo il DMA, quindi l'uso del trigger del timer sarebbe tecnicamente inutile (poiché comunque ogni valore del DHR avviene ad ogni ciclo del for, che non è detto sia perfettamente sincrono con il timer), però per sfizio lo usiamo lo stesso.

### Ricordiamoci del jumper tra PA4 e PA2!

Per prima cosa generiamo allora il seno con l'ausilio della funzione sin() nella libreria <math.h>. Se decidiamo di avere 100 campioni per ogni periodo la formula da usare è:

$$V_{\text{SIN}} = 1.5 + \text{ampiezza} * \sin(2\pi i / 100)$$

dove:

- 1.5 è l'offset da aggiungere per via del fatto che la tensione assume valori da 0 a 3V, quindi non sono ammessi valori negativi;
- l'ampiezza varia quindi tra [-1.5 ; +1.5]
- $i$  varia tra 0 e 99

```
void sin_gen(float ampiezza) {
    float Vsin;
    for(int i=0; i<N; i++){
        Vsin = 1.5 + ampiezza*sin(2*PI*i/N);
        LUT[i] = (short int)(Vsin*4095.0/3.0); //codifica dei valori della sinusoide
    }
}
```

Poiché il DAC richiede in ingresso non una tensione, ma un codice, andiamo a creare un vettore LUT (Look-up Table) (globale per la live watch) in cui memorizziamo le 100 codifiche delle tensioni del seno. Ogni elemento del vettore deve essere **short int** per via del fatto che DAC e ADC lavorano a 16 bit (ricordiamo che però il codice arriva ad occupare 12 bit).

Dopo aver abilitato i clock delle varie periferiche, ricordiamo di impostare PA4 (del DAC) e PA2 (dell'ADC) come porte analogiche, dopodiché passiamo alla configurazione del timer, del DAC e dell'ADC.

Per quanto riguarda l'**ADC**, impostiamo **CONT=0**, perché ogni conversione deve avvenire dopo la generazione di un campione e quindi setteremo **ADSTART** ad ogni ciclo del for. Se volessimo usare il trigger del timer anche sull'ADC allora basterebbe mettere un unico **ADSTART** prima del for e settare **CONT=1**, poiché sarebbe il timer a preoccuparsi di scandire le conversioni.

```
void setup_ADC(){
    // Abilitazione regolatore di tensione
    ADC1->CR &= ~ADC_CR_ADVREGEN_1; //ADVREGEN = 10->00
    ADC1->CR |= ADC_CR_ADVREGEN_0; //ADVREGEN = 00->01
    for(int i=0; i<1000; i++); //attesa di 10 µs

    // Configurazione clock
    ADC1_2->CCR |= ADC12_CCR_CKMODE_0; //CKMODE=01, clock del bus AHB

    // Calibrazione ADC
    ADC1->CR |= ADC_CR_ADCAL; //ADCAL=1, avvio calibrazione
    while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL); //attesa ADCAL=0

    // Abilitazione ADC
    ADC1->CR |= ADC_CR_ADEN; //ADEN=1, abilitazione ADC
    while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD); //attesa ADRDY=1

    // Configurazione ADC
    ADC1->CFGCR &= ~ADC_CFGCR_CONT; //CONT=0, conversione singola
    ADC1->SQR1 = (3<<6); //SQ1=00011: canale 3 (PA2)
    ADC1->SQR1 &= ~ADC_SQR1_L; //L=0: 1 conv.
    ADC1->SMPCR1 |= ADC_SMPCR1_SMP3; //SMP3=111, 601.5 CK
}
```

```
void setup_DAC(){
    DAC->CR |= DAC_CR_TEN1; //abilitazione trigger
    DAC->CR |= DAC_CR_TSSEL1_2; //TSEL=100: trigger TIM2
    DAC->CR |= DAC_CR_EN1; //abilitazione canale 1
}
```

Per quanto riguarda il **DAC**, bisogna abilitare il trigger con **TEN = 1** e selezionare il trigger **TRGO** del **TIM2** con **TSEL = 100**.

Per quanto riguarda il **timer**, si sceglie il tempo di clock, ad esempio mezzo secondo (che corrisponde a 36 milioni

di colpi di clock se la frequenza è impostata a 72 MHz, tramite i soli 2 file da aggiungere al progetto: "startup\_stm32f303xc.s" e "system\_stm32f30x.c") e si imposta **MMS (Master Mode Selection)** a 010, cioè imponendo al timer di generare dei trigger TRGO ad ogni evento di update.

```
void setup_TIM(){
    TIM2->ARR = 36000000; //0.5 s
    TIM2->CR2 = TIM_CR2_MMS_1; //MMS = 010 per far scattare il trigger
}
```

A questo punto realizziamo il ciclo in cui:

- Aggiorniamo il DHR
- Aspettiamo che il timer copi il DHR nel DOR
- Aspettiamo che il DAC generi la tensione
- Convertiamo tale valore
- Mettiamo il risultato in un secondo vettore che chiamiamo LUT2 (globale per la live watch)

```
for(int i=0; i<N; i++){
    // Scrittura codice nel DHR
    DAC1->DHR1R1 = LUT[i];
    while((TIM2->SR & TIM_SR_UIF) != TIM_SR_UIF); //attesa UIF=1
    TIM2->SR &= ~TIM_SR_UIF; //UIF=0
    for(int j=0; j<1000; j++); //attesa generazione di tensione

    // Conversione
    ADC1->CR |= ADC_CR_ADSTART; //ADSTART=1, avvio conversione
    while((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC); //attesa EOC=1

    // Lettura risultato
    LUT2[i] = ADC1->DR;

    // Disabilitazione ADC, DAC, TIM
    disabilitazione_periferiche();
}
```

```

#include <math.h>
#include <stm32f30x.h>
#define N 100
#define PI 3.14159265

short int LUT[N];
short int LUT2[N];

void sin_gen(float);
void abilitazione_periferiche();
void disabilitazione_periferiche();
void setup_ADC();
void setup_DAC();
void setup_TIM();

void main() {
    // Generazione della sinusoide
    sin_gen(1.3);

    // Abilitazione DAC1, ADC12, GPIOA, TIM2
    abilitazione_periferiche();

    // PA4 e PA2 in modalità analogica
    GPIOA->MODER |= GPIO_MODER_MODER4;
    GPIOA->MODER |= GPIO_MODER_MODER2;

    // setup di DAC1, ADC12, TIM2
    setup_DAC();
    setup_ADC();
    setup_TIM();

    TIM2->CNT = 0;                                //azzero il conteggio
    TIM2->CR1 = TIM_CR1_CEN;                      //avvio conteggio

    for(int i=0; i<N; i++){
        // Scrittura codice nel DHR
        DAC1->DHR12R1 = LUT[i];
        while((TIM2->SR & TIM_SR UIF) != TIM_SR UIF);      //attesa UIF=1
        TIM2->SR &= ~TIM_SR UIF;                          //UIF=0
        for(int j=0; j<1000; j++);                      //attesa generazione di tensione

        // Conversione
        ADC1->CR |= ADC_CR_ADSTART;                  //ADSTART=1, avvio conversione
        while((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC); //attesa EOC=1

        // Lettura risultato
        LUT2[i] = ADC1->DR;
    }

    // Disabilitazione ADC, DAC, TIM
    disabilitazione_periferiche();

    while(1);
}

```

Generazione sinusoide (senza DMA)

```

//-----
void sin_gen(float ampiezza){
    float Vsin;
    for(int i=0; i<N; i++){
        Vsin = 1.5 + ampiezza*sin(2*PI*i/N);
        LUT[i] = (short int)(Vsin*4095.0/3.0); //codifica dei valori della sinusoide
    }
}

void abilitazione_periferiche(){
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //abilitazione GPIOA
    RCC->AHBENR |= RCC_AHBENR_ADC12EN; //abilitazione ADC12
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; //abilitazione TIM2
    RCC->APB1ENR |= RCC_APB1ENR_DAC1EN; //abilitazione DAC1
}

void disabilitazione_periferiche(){
    ADC1->CR |= ADC_CR_ADDIS; //ADDIS=1
    DAC1->CR &= ~DAC_CR_EN1; //EN1=0
    TIM2->CR1 &= ~TIM_CR1_CEN; //CEN=0
}

void setup_ADC(){
    // Abilitazione regolatore di tensione
    ADC1->CR &= ~ADC_CR_ADVREGEN_1; //ADVREGEN = 10->00
    ADC1->CR |= ADC_CR_ADVREGEN_0; //ADVREGEN = 00->01
    for(int i=0; i<1000; i++); //attesa di 10 us

    // Configurazione clock
    ADC1_2->CCR |= ADC12_CCR_CKMODE_0; //CKMODE=01, clock del bus AHB

    // Calibrazione ADC
    ADC1->CR |= ADC_CR_ADCAL; //ADCAL=1, avvio calibrazione
    while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL); //attesa ADCAL=0

    // Abilitazione ADC
    ADC1->CR |= ADC_CR_ADEN; //ADEN=1, abilitazione ADC
    while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD); //attesa ADRDY=1

    // Configurazione ADC
    ADC1->CFGR &= ~ADC_CFGR_CONT; //CONT=0, conversione singola
    ADC1->SQR1 = (3<<6); //SQ1=00011: canale 3 (PA2)
    ADC1->SQR1 &= ~ADC_SQR1_L; //L=0: 1 conv.
}

void setup_DAC(){
    DAC->CR |= DAC_CR_TEN1; //abilitazione trigger
    DAC->CR |= DAC_CR_TSEL1_2; //TSEL=100: trigger TIM2
    DAC->CR |= DAC_CR_EN1; //abilitazione canale 1
}

void setup_TIM(){
    TIM2->ARR = 36000000; //0.5 s
    TIM2->CR2 = TIM_CR2_MMS_1; //MMS = 010 per far scattare il trigger
}

```

*Picco a ¼ del periodo (25)  
Mínimo a ¾ del periodo (75)*

Expression	Value
LUT <array>	
[0] 2047	2049
[1] 2158	2160
[2] 2269	2271
[3] 2380	2380
[4] 2488	2490
[5] 2595	2596
[6] 2700	2702
[7] 2803	2806
[8] 2902	2904
[9] 2998	3000
[10] 3090	3093
[11] 3178	3181
[12] 3262	3266
[13] 3341	3345
[14] 3414	3416
[15] 3483	3486
[16] 3545	3549
[17] 3602	3604
[18] 3653	3656
[19] 3697	3700
[20] 3735	3738
[21] 3766	3770
[22] 3790	3794
[23] 3808	3811
[24] 3818	3822
[25] 3821	3824
[26] 3818	3822
[27] 3808	3810
[28] 3790	3792
[29] 3766	3768
[30] 3735	3738
[31] 3697	3701
[32] 3653	3657
[33] 3602	3605
[34] 3545	3548
[35] 3483	3487
[36] 3414	3418
[37] 3341	3345
[38] 3262	3266
[39] 3178	3180
[40] 3090	3093
[41] 2998	3000
[42] 2902	2904
[43] 2803	2804
[44] 2700	2702
[45] 2595	2596
[46] 2488	2489
[47] 2380	2381
[48] 2269	2270
[49] 2158	2160
[50] 2047	2050
[51] 1936	1938
[52] 1825	1826
[53] 1714	1715
[54] 1606	1607
[55] 1499	1501
[56] 1394	1395
[57] 1291	1292
[58] 1192	1193
[59] 1096	1098
[60] 1004	1006
[61] 916	917
[62] 832	832
[63] 753	754
[64] 680	680
[65] 611	612
[66] 549	550
[67] 492	492
[68] 441	441
[69] 397	397
[70] 359	360
[71] 328	329
[72] 304	305
[73] 286	287
[74] 276	277
[75] 273	274
[76] 276	276
[77] 286	287
[78] 304	305
[79] 328	327
[80] 359	359
[81] 397	397
[82] 441	442
[83] 492	491
[84] 549	548
[85] 611	611
[86] 680	680
[87] 753	753
[88] 832	832
[89] 916	916
[90] 1004	1005
[91] 1096	1096
[92] 1192	1193
[93] 1291	1292
[94] 1394	1394
[95] 1499	1500
[96] 1606	1608
[97] 1714	1715
[98] 1825	1827
[99] 1936	1937

# 9. Interruzioni (EXTI)

## 9.1 Interruzioni delle GPIO

Alcuni bit dei registri ISER dell'NVIC corrispondono a interruzioni di tipo EXTI (*Extended Interrupts and Events Controller*).

L'EXTI gestisce fino a 36 interrupts.

In particolare, 16 delle 28 linee esterne sono collegate ai pin:

- EXTI0: PA0, ..., PF0
- EXTI1: PA1, ..., PF1
- ...
- EXTI15: PA15, ..., PE15

Position	Priority	Type of priority	Acronym	Description
5	12	settable	RCC	RCC global interrupt
6	13	settable	EXTI0	EXTI Line0 interrupt
7	14	settable	EXTI1	EXTI Line1 interrupt
8	15	settable	EXTI2 and TSC	EXTI Line2 and Touch sensing interrupts
9	16	settable	EXTI3	EXTI Line3
10	17	settable	EXTI4	EXTI Line4
11	18	settable	DMA1_CH1	DMA1 channel 1 interrupt

Per scegliere la linea (0 ... 15) si seleziona quindi il corrispondente EXTI<sub>i</sub>, mentre per selezionare la porta (A ... F) si ricorre ai registri SYSCFG\_EXTICR[x].

Affinché il controller attivi la routine relativa alla linea scelta, l'interrupt deve essere smascherata modificando i bit del registro IMR (*Interrupt Mask Register*).

Per settare una linea come fonte di interrupt hardware:

- Configurare il bit corrispondente nel registro EXTI\_IMR
- Configurare il fronte di salita e/o discesa nei registri EXTI\_RTSR ed EXTI\_FTSR
- Configurare i bit che controllano il canale NVIC mappato verso l'EXTI

Per settare una linea come fonte di interrupt software:

- Configurare il bit corrispondente nel registro EXTI\_IMR
- generare l'evento/interruzione settando il bit SWIER nel registro EXTI\_SWIER

Nei registri delle Pending Request (EXTI\_PR) vengono attivati i bit relativi a una determinata linea per segnalare che l'interrupt è stato servito su quella linea. Tale bit va pulito all'interno della ISR scrivendo '1'.

**Nota:** se si attiva l'interrupt sia sul fronte di salita che di discesa, possiamo ricorrere ad un flag per riconoscere il fronte. Ad esempio inizializziamo il flag a 1 e inseriamo nella funzione handler un if(flag) / else. In questo modo quando si preme, ad esempio, il pulsante, scatta l'interruzione ed esegue il codice nell'if (che si occupa anche di cambiare il valore del flag); quando si lascia il pulsante scatta un'altra interruzione che stavolta porta al codice dell'else, essendo cambiato il flag.

```
int flag = 1;

if(flag){           //fronte di salita
...
flag = 0;
}
else{              //fronte di discesa
...
flag = 1;
}
```

## 9.2 Registri utili delle interruzioni EXTI

I principali registri sono:

- **SYSCFG\_EXTICR[x] (EXTI Configuration Registers)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw												

- **EXTIx (EXTI della linea x)**
  - 000: Porta A
  - 001: Porta B
  - 010: Porta C
  - 011: Porta E
  - 100: Porta F

- **EXTI\_IMR (Interrupt Mask Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MR31	MR30	MR29	MR28	MR27	MR26	MR25	MR24	MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
rw															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw															

- **MRx (Mask Request)**
  - 1: l'interrupt della linea x è smascherata

- **EXTI\_RTSR e EXTI\_FTSR** (*Rising/Falling Trigger Selection Register*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TR31	TR30	TR29	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	TR19	TR18	TR17	TR16
rw	rw	rw							rw						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw															

- **TRx** (*Trigger*)
  - 1: trigger sul fronte di salita/discesa abilitato

- **EXTI\_SWIER** (*Software Interrupt Event Register*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SWIER 31	SWIER 30	SWIER 29	Res.	Res.	Res.	Res.	Res.	Res.	SWIER 22	SWIER 21	SWIER 20	SWIER 19	SWIER 18	SWIER 17	SWIER 16
rw	rw	rw							rw						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWIER 15	SWIER 14	SWIER 13	SWIER 12	SWIER 11	SWIER 10	SWIER 9	SWIER 8	SWIER 7	SWIER 6	SWIER 5	SWIER 4	SWIER 3	SWIER 2	SWIER 1	SWIER 0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **SWIERx** (*Software Interrupt Event*)
  - 0: il bit è pulito dai registri PR (scrivendo ‘1’)
  - 1: evento

- **EXTI\_PR** (*Pending Register*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PR31	PR30	PR29	Res.	Res.	Res.	Res.	Res.	Res.	PR22	PR21	PR20	PR19	PR18	PR17	PR16
rc_w1	rc_w1	rc_w1							rc_w1						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1															

- **PRx** (*Pending Request*)
  - 1: l'interruzione è stata servita (si pulisce riscrivendo ‘1’)

## 9.3 Progetto 12: conversione alla pressione di USER

Vogliamo riscrivere l'esercizio del DAC e ADC in serie, ma questa volta la conversione avviene alla pressione del pulsante USER. La soluzione più semplice è quella di scrivere ADSTART appena è premuto il pulsante, quindi dopo un while che controlla la pressione del pulsante. Tuttavia vogliamo esercitarcì ad usare le interruzioni, quindi useremo le EXTI.

Per prima cosa aggiungiamo i 2 file NECESSARI per l'utilizzo delle interruzioni (ma che per comodità conviene aggiungere ad ogni progetto) "startup\_stm32f303xc.s" e "system\_stm32f30x.c".

Abilitiamo poi tutto ciò che occorre (GPIOA, DAC, ADC, SYSCFG).

**Attenzione:** non vogliamo che venga convertita la linea del pulsante, ma la linea del DAC alla pressione del pulsante. Quindi ricordiamoci del jumper tra i pin PA4 e PA2.

Avremo quindi le linee PA0 digitale e PA2 e PA4 analogiche.

```
// Configurazione Interrupt
SYSCFG->EXTICR[0] &= ~7;
EXTI->IMR |= EXTI_IMR_MR0;
EXTI->RTSR |= EXTI_RTSR_TR0;
NVIC->ISER[0] |= (1<<6);                                //EXTI0=000: porta A
                                                               //maschera disabilitata per la linea 0
                                                               //fronte di salita
                                                               //NVIC serve EXTI0

void EXTI0_IRQHandler() {
    EXTI->PR |= EXTI_PR_PR0;                                //cancellazione Pending Request

    // Conversione
    ADC1->CR |= ADC_CR_ADSTART;                           //ADSTART=1, avvio conversione
    while((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC)     //attesa EOC=1

    // Risultato (ADC)
    codice_out = ADC1->DR;
    tensione_in = codice_out*(3.0/4096.0);

    printf("ADC\n");
    printf("ingresso: %f V\n", tensione_in);
    printf("uscita: %d\n\n", codice_out);
}
```

Vogliamo che la linea PA0 sia configurata come fonte di interrupt perciò:

- settiamo PA0 come input digitale
- selezioniamo la porta A nel registro SYSCFG\_EXTICR[0]
- selezioniamo la linea 0 nel registro IMR
- selezioniamo il fronte di salita nel registro RTSR
- impostiamo NVIC per servire EXTI0 (bit 6 di ISER[0])
- Implementare la funzione EXTI0\_IRQHandler:
  - Cancellare la Pending Request nel registro PR1
  - far partire ADC
  - mostrare il risultato

6 | 13 | settable | EXTI0 | EXTI Line0 interrupt

```

#include <stdio.h>
#include <stm32f30x.h>

// variabili del DAC
int codice_in = 2017; //codice in ingresso [0; 4095]
float tensione_out; //stima della tensione generata dal DAC

// variabili dell'ADC
float tensione_in; //stima della tensione letta dall'ADC
int codice_out; //risultato dell'ADC

void abilitazione_periferiche();
void disabilitazione_periferiche();
void setup_ADC();
void setup_DAC();

void main(){
    // Abilitazione DAC1, ADC12, GPIOA, SYSCFG
    abilitazione_periferiche();

    // PA4, PA2, PA0
    GPIOA->MODER |= GPIO_MODER_MODER4; //analogico
    GPIOA->MODER |= GPIO_MODER_MODER2; //analogico
    GPIOA->MODER &= ~GPIO_MODER_MODER0; //digitale (input)

    // Setup di DAC e ADC
    setup_DAC();
    setup_ADC();

    // Configurazione Interrupt
    SYSCFG->EXTICR[0] &= ~7; //EXTI0=000: porta A
    EXTI->IMR |= EXTI_IMR_MR0; //maschera disabilitata per la linea 0
    EXTI->RTSR |= EXTI_RTSR_TR0; //fronte di salita
    NVIC->ISER[0] |= (1<<6); //NVIC serve EXTI0

    // Risultato (DAC)
    tensione_out = codice_in * (3.0/4095.0);

    printf("DAC\n");
    printf("ingresso: %d\n", codice_in);
    printf("uscita: %f V\n\n", tensione_out);
    printf("-----\n\n");

    while(1);
}

```

Conversione alla pressione del pulsante USER

ingresso: 1.478760 V  
uscita: 2019

ADC  
ingresso: 1.478027 V  
uscita: 2018

ADC  
ingresso: 1.478760 V  
uscita: 2019

ADC  
ingresso: 1.479492 V  
uscita: 2020

```

void abilitazione_periferiche(){
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //abilitazione GPIOA
    RCC->AHBENR |= RCC_AHBENR_ADC12EN; //abilitazione ADC12
    RCC->APB1ENR |= RCC_APB1ENR_DAC1EN; //abilitazione DAC1
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; //abilitazione SYSCFG
}

void setup_DAC(){
    DAC1->CR |= DAC_CR_EN1; //abilitazione canale 1
    DAC1->DHR12R1 = codice_in; //scrittura codice nel DHR
    for(int i=0; i<1000; i++); //attesa generazione di tensione
}

void setup_ADC(){
    // Abilitazione regolatore di tensione
    ADC1->CR &= ~ADC_CR_ADVREGEN_1; //ADVREGEN = 10->00
    ADC1->CR |= ADC_CR_ADVREGEN_0; //ADVREGEN = 00->01
    for(int i=0; i<1000; i++); //attesa di 10 µs

    // Configurazione clock
    ADC1_2->CCR |= ADC12_CCR_CKMODE_0; //CKMODE=01, clock del bus AHB

    // Calibrazione ADC
    ADC1->CR |= ADC_CR_ADCAL; //ADCAL=1, avvio calibrazione
    while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL); //attesa ADCAL=0

    // Abilitazione ADC
    ADC1->CR |= ADC_CR_ADEN; //ADEN=1, abilitazione ADC
    while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD); //attesa ADRDY=1

    // Configurazione ADC
    ADC1->CFGGR &= ~ADC_CFGGR_CONT; //CONT=0, conversione singola
    ADC1->SQR1 = (3<<6); //SQ1=00011: canale 3 (PA2)
    ADC1->SQR1 &= ~ADC_SQR1_L; //L=0: 1 conv.
    ADC1->SMPR1 |= ADC_SMPR1_SMP3; //SMP3=111, 601.5 CK
}

void EXTI0_IRQHandler(){
    EXTI->PR |= EXTI_PR_PR0; //cancellazione Pending Request

    // Conversione
    ADC1->CR |= ADC_CR_ADSTART; //ADSTART=1, avvio conversione
    while((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC); //attesa EOC=1

    // Risultato (ADC)
    codice_out = ADC1->DR;
    tensione_in = codice_out*(3.0/4096.0);

    printf("ADC\n");
    printf("ingresso: %f V\n", tensione_in);
    printf("uscita: %d\n\n", codice_out);
}

```

# 10. DMA

## 10.1 Descrizione Generale

Il **Direct Memory Access (DMA)** permette spostamenti di dati senza l'intervento della CPU.

Lo schedino è dotato di 2 controller DMA (DMA1 e DMA2) che hanno rispettivamente 7 e 5 canali e sono mappati sul bus AHB.

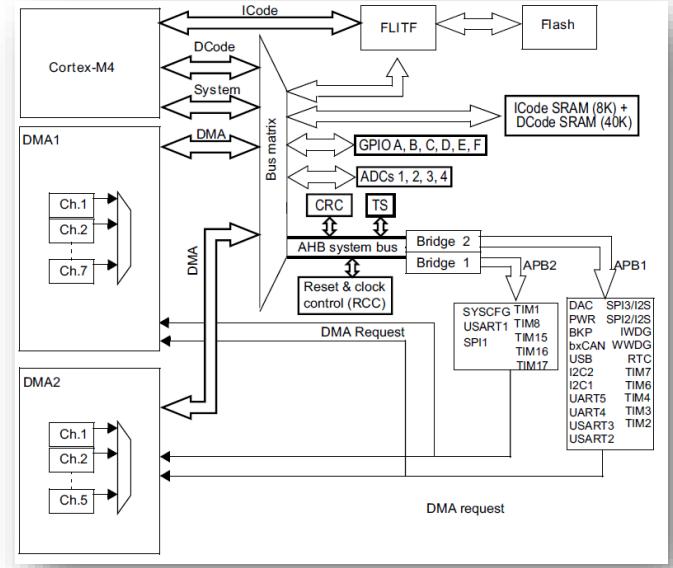
I controller DMA condividono il bus di sistema con il processore (Cortex-M4) per poter comunicare direttamente con la memoria e le periferiche.

Il DMA può operare in 2 modalità:

- memoria ↔ memoria
- memoria ↔ periferica

Nel secondo caso, il trasferimento dei dati avviene in seguito ad un evento della periferica.

Nel caso del DAC, ad esempio, l'occorrenza di un trigger hardware, oltre a generare il passaggio dei dati dal DHR al DOR, genera anche una richiesta al DMA, se DMAEN è settato.



La dimensione dei dati della periferica e della memoria può essere modificata tramite i bit PSIZE e MSIZE.

I registri CPAR e CMAR vanno riempiti con l'indirizzo della periferica e della memoria in questione. Tramite i bit MINC e PINC si può inoltre scegliere di incrementare automaticamente il puntatore memoria/periferica ad ogni trasferimento.

Il numero di trasferimenti da effettuare va specificato nel registro CNDTR, il cui valore è poi decrementato di volta in volta. Se è selezionata la modalità circolare, alla fine delle operazioni CNDTR assume di nuovo il valore iniziale e tutti i dati vengono ritrasferiti, e così via ciclicamente.

Per lavorare con il DMA occorre:

- abilitare il clock del DMA in RCC\_AHBENR
- effettuare il remapping del canale in SYSCFG\_CFGR1
- inserire l'indirizzo della periferica in CPAR
- inserire l'indirizzo della locazione di memoria in CMAR
- specificare il numero di trasferimenti in CNDTR
- configurazioni varie (direzione, modalità circolare, incremento automatico, dimensione dati...)
- attivazione canale con EN

Ovviamente la periferica deve aver abilitato il canale DMA.

## 10.2 Registri utili del DMA

I principali registri sono:

- **DMA\_ISR (Interrupt Status Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5
				r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- TEIFx (*Transfer Error Interrupt Flag*)
- HTIFx (*Half Transfer Interrupt Flag*)
- TCIFx (*Transfer Complete Interrupt Flag*)
- GIFx (*Global Interrupt Flag*)
  - 1: è avvenuto un evento di TE, HT o TC

**Nota:** i flag vanno resettati nel registro DMA\_IFCR

- **DMA\_IFCR (Interrupt Flag Clear Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	CTEIF7	CHTIF7	CTCIF7	CGIF7	CTEIF6	CHTIF6	CTCIF6	CGIF6	CTEIF5	CHTIF5	CTCIF5	CGIF5
				w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTEIF4	CHTIF4	CTCIF4	CGIF4	CTEIF3	CHTIF3	CTCIF3	CGIF3	CTEIF2	CHTIF2	CTCIF2	CGIF2	CTEIF1	CHTIF1	CTCIF1	CGIF1
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

- CTEIFx (*Clear Transfer Error Interrupt Flag*)
- CHTIFx (*Clear Half Transfer Interrupt Flag*)
- CTCIFx (*Clear Transfer Complete Interrupt Flag*)
- CGIFx (*Clear Global Interrupt Flag*)

- **DMA\_CNDTRx (Channel x Number of Data Transfers Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NDT															
rw															

- **DMA\_CCRx** (*Channel x Configuration Register*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	MEM2 MEM	PL[1:0]		MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN
		RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

- **PL (Priority Level)**
  - 00: Low
  - 01: Medium
  - 10: High
  - 11: Very High
- **MSIZE / PSIZE (Memory / Peripheral Size)**
  - 00, 01, 10: dati di 8, 16, 32 bit
- **MINC / PINC (Memory / Peripheral Increment)**
  - 1: incremento automatico
- **CIRC (Circular Mode)**
  - 1: modalità circolare
- **DIR (Direction)**
  - 0: periferica → memoria
  - 1: memoria → periferica
- **EN (Enable)**
  - 1: canale x abilitato

- **DMA\_CPARx / DMA\_CMARx** (*Channel x Peripheral/Memory Address Register*)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PA																															
RW																															

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MA																															
RW																															

## 10.3 Progetto 13: generazione sinusoida con DMA

Esercizio simile al precedente, ma stavolta sfruttiamo il DMA.

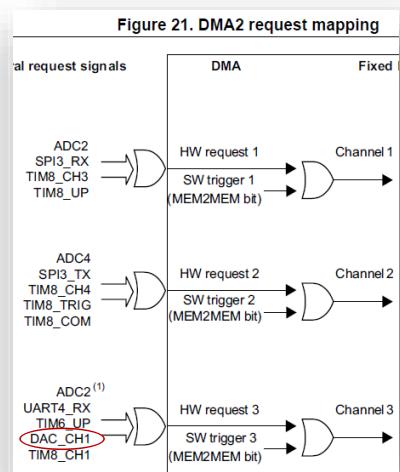
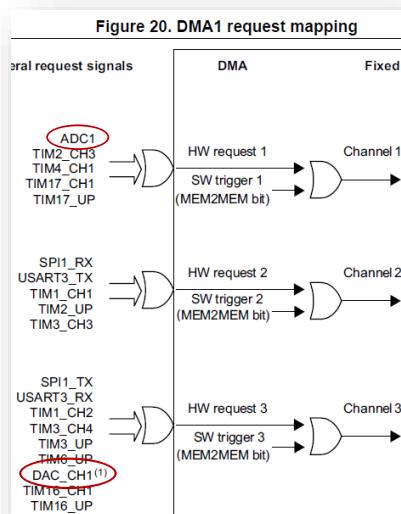
In questo caso il DMA è molto conveniente, perché ci permette anche di incrementare automaticamente l'indirizzo del valore da passare al DAC.

Poiché una volta avviato il DAC in modalità DMA l'incremento è automatico, abbiamo un duplice vantaggio:

- non abbiamo bisogno di cicli for
- generazione e acquisizione sono indipendenti, perché non devono più trovarsi entrambi in uno stesso ciclo

Proviamo quindi a generare infiniti periodi con il DAC e ad acquisirne, per esempio, solo uno. Sfrutteremo quindi la modalità circolare (CIRC) per il DMA, così il vettore ricomincia ogni volta che giunge all'ultimo elemento.

Possiamo risolvere l'esercizio usando il DMA per entrambi i convertitori: il DAC prenderà automaticamente i valori dal vettore LUT, l'ADC inserirà automaticamente i valori convertiti nel vettore LUT2.



Per l'ADC1 dobbiamo per forza ricorrere al canale 1 del DMA1, mentre per il DAC possiamo usare il canale 3 del DMA1 o del DMA2.

**Attenzione** però, per usare il canale 3 del DMA1 in favore del DAC bisogna effettuare un **remapping** del canale 3 nel modo seguente:

- abilitare il clock del SYSCFG nel registro RCC\_APB2ENR
- settare il bit TIM6\_DAC1\_DMA\_RMP nel registro SYSCFG\_CFGR1

```
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
SYSCFG->CFGR1 |= SYSCFG_CFGR1_TIM6DAC1;
```

Peripherals	Channel 1	Channel 2	Channel 3
ADC	ADC1		
SPI		SPI1_RX	SPI1_TX
USART		USART3_TX	USART3_RX
I2C			
TIM1		TIM1_CH1	TIM1_CH2
TIM2	TIM2_CH3	TIM2_UP	
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP
TIM4	TIM4_CH1		
TIM6 / DAC			TIM6_UP DAC_CH1 (1)

Per non complicarci la vita usiamo il DMA2 per il DAC, così evitiamo il remapping.

Abilitiamo quindi tutte le periferiche che ci occorrono e generiamo il vettore con le codifiche della sinusoide, che non cambia rispetto all'esercizio di prima.

```
void sin_gen(float ampiezza) {
    float Vsin;
    for(int i=0; i<N; i++){
        Vsin = 1.5 + ampiezza*sin(2*PI*i/N);
        LUT[i] = (short int)(Vsin*4095.0/3.0);
    }
}

void abilitazione_periferiche(){
    RCC->AHBENR |= RCC_AHBENR_DMA1EN; //abilitazione DMA1
    RCC->AHBENR |= RCC_AHBENR_DMA2EN; //abilitazione DMA2
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //abilitazione GPIOA
    RCC->AHBENR |= RCC_AHBENR_ADC12EN; //abilitazione ADC12
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; //abilitazione TIM2
    RCC->APB1ENR |= RCC_APB1ENR_DAC1EN; //abilitazione DAC1
}
```

Per quanto riguarda il DMA dell'ADC (DMA1 canale 1):

- inseriamo in CPAR l'indirizzo del DR
- inseriamo in CMAR l'indirizzo del LUT2
- indichiamo in CNDTR il numero di trasferimenti
- impostiamo le dimensioni di memoria e periferica a 16 bit
- impostiamo l'incremento automatico per il vettore (quindi lato memoria)
- impostiamo la direzione periferica -> memoria (DIR = 0)
- infine abilitiamo il canale 1

```
void setup_DMA1() {
    DMA1_Channel1->CPAR = (uint32_t)&ADC1->DR; //indirizzo periferica
    DMA1_Channel1->CMAR = (uint32_t)LUT2; //indirizzo memoria
    DMA1_Channel1->CNDTR = N; //numero trasferimenti
    DMA1_Channel1->CCR |= DMA_CCR_MSIZE_0; //MSIZE = 16 bit
    DMA1_Channel1->CCR |= DMA_CCR_PSIZE_0; //PSIZE = 16 bit
    DMA1_Channel1->CCR |= DMA_CCR_MINC; //incremento indirizzo memoria
    DMA1_Channel1->CCR &= ~DMA_CCR_DIR; //DIR=0: periferica -> memoria
    DMA1_Channel1->CCR |= DMA_CCR_EN; //abilitazione canale 1
}
```

Per quanto riguarda il DMA del DAC (DMA2 canale 3):

- inseriamo in CPAR l'indirizzo del DHR
- inseriamo in CMAR l'indirizzo del LUT
- indichiamo in CNDTR il numero di trasferimenti
- impostiamo le dimensioni di memoria e periferica a 16 bit
- impostiamo l'incremento automatico per il vettore (quindi lato memoria)
- impostiamo la modalità circolare per generare infiniti periodi (CIRC=1)
- impostiamo la direzione memoria -> periferica (DIR = 1)
- infine abilitiamo il canale 3

```
void setup_DMA2() {
    DMA2_Channel3->CPAR = (uint32_t)&DAC1->DHR12R1; //indirizzo periferica
    DMA2_Channel3->CMAR = (uint32_t)LUT; //indirizzo memoria
    DMA2_Channel3->CNDTR = N; //numero trasferimenti
    DMA2_Channel3->CCR |= DMA_CCR_MSIZE_0; //MSIZE = 16 bit
    DMA2_Channel3->CCR |= DMA_CCR_PSIZE_0; //PSIZE = 16 bit
    DMA2_Channel3->CCR |= DMA_CCR_MINC; //incremento indirizzo memoria
    DMA2_Channel3->CCR |= DMA_CCR_CIRC; //modalità circolare
    DMA2_Channel3->CCR |= DMA_CCR_DIR; //DIR=1: memoria -> periferica
    DMA2_Channel3->CCR |= DMA_CCR_EN; //abilitazione canale 3
}
```

Passiamo alla configurazione dell'ADC. Per far "scattare" il DMA ogni tot tempo, oltre al DMA abilitiamo anche il trigger del timer sull'ADC.

In questo modo quando si verifica l'evento di update del timer:

- la tensione viene convertita (se ADSTART è stato in precedenza settato a 1)
- il contenuto del DR viene messo nel vettore dei risultati LUT2

Purtroppo quei gran geniacchi che hanno progettato il convertitore hanno fatto in modo che venga trasmesso il contenuto "vecchio" del DR, non il valore appena convertito. Per questo il primo valore convertito potrebbe essere in realtà un valore "fasullo", essendo il DR vuoto all'inizio (ma per qualche fenomeno anomalo a me non succede, non ho capito perché).

Bisogna quindi:

- Abilitare la modalità DMA con DMAEN = 1
- Abilitare i trigger con EXTEN = 01 (sul fronte di salita)
- Selezionare il trigger TRGO del TIM2 con EXTSEL = 1011

Name	Source	Type	EXTSEL[3:0]
EXT11	TIM2_TRGO event	Internal signal from on chip timers	1011
EXT12	TIM4_TRGO event	Internal signal from on chip timers	1100
EXT13	TIM2_TRGO event	Internal signal from on chip timers	1101

Tocca ora a Donald DAC. Per il papero il discorso è simile: vogliamo attivare ancora una volta il DMA e il trigger del timer:

- Abilitare la modalità DMA con DMAEN = 1
- Abilitare i trigger con TEN = 1
- Selezionare il trigger TRGO del TIM2 con TSEL = 100

Ancora una volta i geniacchi hanno fatto in modo che venga convertito il contenuto "vecchio" del DHR e non il valore appena trasferito con il DMA. Per questo il primo valore generato è in realtà un valore "fasullo", essendo il DHR vuoto all'inizio. Ma questo poco importa, basta far partire l'ADC un po' più tardi.

Per il timer stesso  
discorso dell'altra volta.

```
void setup_TIM() {
    TIM2->ARR = 36000000; //0.5 s
    TIM2->CR2 = TIM_CR2_MMS_1; //MMS = 010 per far scattare il trigger
}
```

Nel main, dopo le varie abilitazioni e configurazioni, ci tocca avviare il timer. Questo fa scattare inizialmente solo il DMA del DAC, che comincia a generare il suo gran bel seno. Attendiamo un po', dopodiché attiviamo l'ADC con il bit ADSTART. Non resta che confrontare i risultati osservando LUT e LUT2 in live watch. I valori letti saranno sfasati rispetto a quelli iniziali per via del ritardo che abbiamo imposto all'ADC.

```

#include <math.h>
#include <stm32f30x.h>
#define N 100
#define PI 3.14159265

short int LUT[N];
short int LUT2[N];

void sin_gen(float);
void abilitazione_periferiche();
void disabilitazione_periferiche();
void setup_DMA1();
void setup_DMA2();
void setup_ADC();
void setup_DAC();
void setup_TIM2();
void setup_TIM3();

void main(){

    // Generazione della sinusode
    sin_gen(1.3);

    // Abilitazione DMA1, DMA2, DAC1, ADC12, GPIOA, TIM2
    abilitazione_periferiche();

    // PA4 e PA2 in modalità analogica
    GPIOA->MODER |= GPIO_MODER_MODER4;
    GPIOA->MODER |= GPIO_MODER_MODER2;

    // setup DMA1, DMA2, DAC1, ADC12, TIM2
    setup_DMA1();
    setup_DMA2();
    setup_DAC();
    setup_ADC();
    setup_TIM2();

    TIM2->CNT = 0;                                //azzerio il conteggio
    TIM2->CR1 = TIM_CR1_CEN;                      //avvio conteggio

    for(int i=0; i<10000000; i++);
        ADC1->CR |= ADC_CR_ADSTART;               //avvio conversione

    // prima di disabilitare tutto aspettiamo che l'ADC abbia convertito
    while((DMA1->ISR & DMA_ISR_TCIF3) != DMA_ISR_TCIF3); //attesa trasferimento
    DMA1->IFCR |= DMA_IFCR_CTCIF3; //abbassa il flag

    // Disabilitazione DMA1, DMA2, ADC, DAC, TIM2
    disabilitazione_periferiche();

    while(1);
}

```

Generazione sinusode (con DMA)

```

void sin_gen(float ampiezza){
    float Vsin;
    for(int i=0; i<N; i++){
        Vsin = 1.5 + ampiezza*sin(2*PI*i/N);
        LUT[i] = (short int)(Vsin*4095.0/3.0); //codifica dei valori della sinusode
    }
}

void abilitazione_periferiche(){
    RCC->AHBENR |= RCC_AHBENR_DMA1EN; //abilitazione DMA1
    RCC->AHBENR |= RCC_AHBENR_DMA2EN; //abilitazione DMA2
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //abilitazione GPIOA
    RCC->AHBENR |= RCC_AHBENR_ADC12EN; //abilitazione ADC12
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; //abilitazione TIM2
    RCC->APB1ENR |= RCC_APB1ENR_DAC1EN; //abilitazione DAC1
}

void disabilitazione_periferiche(){
    ADC1->CR |= ADC_CR_ADDIS;
    DAC1->CR &= ~DAC_CR_EN1;
    TIM2->CR1 &= ~TIM_CR1_CEN;
    DMA1_Channel1->CCR &= ~DMA_CCR_EN;
    DMA2_Channel3->CCR &= ~DMA_CCR_EN;
}

void setup_DMA1(){
    DMA1_Channel1->CPAR = (uint32_t)&ADC1->DR; //indirizzo periferica
    DMA1_Channel1->CMAR = (uint32_t)LUT2; //indirizzo memoria
    DMA1_Channel1->CNTR = N; //numero trasferimenti
    DMA1_Channel1->CCR |= DMA_CCR_MSIZE_0; //MSIZE = 16 bit
    DMA1_Channel1->CCR |= DMA_CCR_PSIZE_0; //PSIZE = 16 bit
    DMA1_Channel1->CCR |= DMA_CCR_MINC; //incremento indirizzo memoria
    DMA1_Channel1->CCR &= ~DMA_CCR_DIR; //DIR=0: periferica -> memoria
    DMA1_Channel1->CCR |= DMA_CCR_EN; //abilitazione canale 1
}

void setup_DMA2(){
    DMA2_Channel3->CPAR = (uint32_t)&DAC1->DHR12R1; //indirizzo periferica
    DMA2_Channel3->CMAR = (uint32_t)LUT; //indirizzo memoria
    DMA2_Channel3->CNTR = N; //numero trasferimenti
    DMA2_Channel3->CCR |= DMA_CCR_MSIZE_0; //MSIZE = 16 bit
    DMA2_Channel3->CCR |= DMA_CCR_PSIZE_0; //PSIZE = 16 bit
    DMA2_Channel3->CCR |= DMA_CCR_MINC; //incremento indirizzo memoria
    DMA2_Channel3->CCR |= DMA_CCR_CIRC; //modalità circolare
    DMA2_Channel3->CCR |= DMA_CCR_DIR; //DIR=1: memoria -> periferica
    DMA2_Channel3->CCR |= DMA_CCR_EN; //abilitazione canale 3
}

void setup_ADC(){
    // Abilitazione regolatore di tensione
    ADC1->CR &= ~ADC_CR_ADVREGEN_1; //ADVREGEN = 10->00
    ADC1->CR |= ADC_CR_ADVREGEN_0; //ADVREGEN = 00->01
    for(int i=0; i<1000; i++); //attesa di 10 us

    // Configurazione clock
    ADC1_2->CCR |= ADC12_CCR_CKMODE_0; //CKMODE=01, clock del bus AHB

    // Calibrazione ADC
    ADC1->CR |= ADC_CR_ADCAL; //ADCAL=1, avvio calibrazione
    while((ADC1->CR & ADC_CR_ADCAL) == ADC_CR_ADCAL); //attesa ADCAL=0

    // Abilitazione ADC
    ADC1->CR |= ADC_CR_ADEN; //ADEN=1, abilitazione ADC
    while((ADC1->ISR & ADC_ISR_ADRD) != ADC_ISR_ADRD); //attesa ADRDY=1

    // Configurazione ADC
    ADC1->CFG0 &= ~ADC_CFG0_CONT; //CONT=0, conversione singola
    ADC1->CFG0 |= ADC_CFG0_DMAEN; //modalità DMA
    ADC1->CFG0 |= ADC_CFG0_EXTEN_0; //Trigger abilitati, fronte di salita
    ADC1->CFG0 |= (11<<6); //EXTSEL=1011: trigger del TIM2
    ADC1->SQR1 = (3<<6); //SQ1=00011: canale 3 (PA2)
    ADC1->SQR1 &= ~ADC_SQR1_L; //L=0: 1 conv.
    ADC1->SMPR1 |= ADC_SMPR1_SMP3; //SMP3=111, 601.5 CK
}

void setup_DAC(){
    DAC->CR |= DAC_CR_TEN1; //abilitazione trigger
    DAC->CR |= DAC_CR_TSSEL1_2; //TSEL=100: trigger TIM2
    DAC->CR |= DAC_CR_EN1; //abilitazione canale 1
    DAC->CR |= DAC_CR_DMAEN1; //modalità DMA
}

void setup_TIM2(){
    TIM2->ARR = 36000000; //0.5 s
    TIM2->CR2 = TIM_CR2_MMS_1; //MMS = 010 per far scattare il trigger
}

```

*Picco a ¼ del periodo (25)  
Mínimo a ¾ del periodo (75)*

Expression	Value
[0] LUT	<array>
[0] [0]	2047
[0] [1]	2158
[0] [2]	2269
[0] [3]	2380
[0] [4]	2488
[0] [5]	2595
[0] [6]	2700
[0] [7]	2803
[0] [8]	2902
[0] [9]	2998
[0] [10]	3090
[0] [11]	3178
[0] [12]	3262
[0] [13]	3341
[0] [14]	3414
[0] [15]	3483
[0] [16]	3545
[0] [17]	3602
[0] [18]	3653
[0] [19]	3697
[0] [20]	3735
[0] [21]	3766
[0] [22]	3790
[0] [23]	3808
[0] [24]	3818
[0] [25]	3821
[0] [26]	3818
[0] [27]	3808
[0] [28]	3790
[0] [29]	3766
[0] [30]	3735
[0] [31]	3697
[0] [32]	3653
[0] [33]	3602
[0] [34]	3545
[0] [35]	3483
[0] [36]	3414
[0] [37]	3341
[0] [38]	3262
[0] [39]	3178
[0] [40]	3090
[0] [41]	2998
[0] [42]	2902
[0] [43]	2803
[0] [44]	2700
[0] [45]	2595
[0] [46]	2488
[0] [47]	2380
[0] [48]	2269
[0] [49]	2158
[0] [50]	2047
[0] [51]	1936
[0] [52]	1825
[0] [53]	1714
[0] [54]	1606
[0] [55]	1499
[0] [56]	1394
[0] [57]	1291
[0] [58]	1192
[0] [59]	1096
[0] [60]	1004
[0] [61]	916
[0] [62]	832
[0] [63]	753
[0] [64]	680
[0] [65]	611
[0] [66]	549
[0] [67]	492
[0] [68]	441
[0] [69]	397
[0] [70]	359
[0] [71]	328
[0] [72]	304
[0] [73]	286
[0] [74]	276
[0] [75]	273
[0] [76]	276
[0] [77]	286
[0] [78]	304
[0] [79]	328
[0] [80]	359
[0] [81]	397
[0] [82]	441
[0] [83]	492
[0] [84]	549
[0] [85]	611
[0] [86]	680
[0] [87]	753
[0] [88]	832
[0] [89]	916
[0] [90]	1004
[0] [91]	1096
[0] [92]	1192
[0] [93]	1291
[0] [94]	1394
[0] [95]	1499
[0] [96]	1606
[0] [97]	1714
[0] [98]	1825
[0] [99]	1936

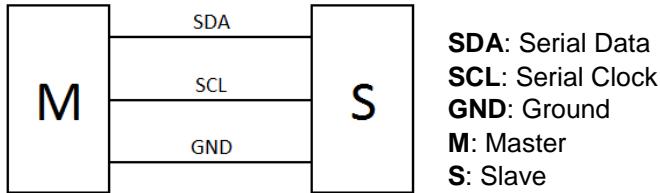
Expression	Value
[0] LUT2	<array>
[0] [0]	2271
[0] [1]	2383
[0] [2]	2490
[0] [3]	2598
[0] [4]	2702
[0] [5]	2805
[0] [6]	2904
[0] [7]	3001
[0] [8]	3094
[0] [9]	3181
[0] [10]	3266
[0] [11]	3344
[0] [12]	3418
[0] [13]	3486
[0] [14]	3547
[0] [15]	3605
[0] [16]	3656
[0] [17]	3700
[0] [18]	3740
[0] [19]	3770
[0] [20]	3794
[0] [21]	3812
[0] [22]	3822
[0] [23]	3823
[0] [24]	3822
[0] [25]	3811
[0] [26]	3795
[0] [27]	3771
[0] [28]	3739
[0] [29]	3700
[0] [30]	3658
[0] [31]	3605
[0] [32]	3548
[0] [33]	3486
[0] [34]	3418
[0] [35]	3346
[0] [36]	3266
[0] [37]	3183
[0] [38]	3094
[0] [39]	3001
[0] [40]	2906
[0] [41]	2804
[0] [42]	2702
[0] [43]	2596
[0] [44]	2491
[0] [45]	2381
[0] [46]	2271
[0] [47]	2159
[0] [48]	2048
[0] [49]	1938
[0] [50]	1827
[0] [51]	1715
[0] [52]	1608
[0] [53]	1500
[0] [54]	1395
[0] [55]	1292
[0] [56]	1193
[0] [57]	1097
[0] [58]	1006
[0] [59]	916
[0] [60]	833
[0] [61]	753
[0] [62]	681
[0] [63]	611
[0] [64]	549
[0] [65]	491
[0] [66]	442
[0] [67]	398
[0] [68]	359
[0] [69]	328
[0] [70]	304
[0] [71]	287
[0] [72]	277
[0] [73]	274
[0] [74]	277
[0] [75]	286
[0] [76]	304
[0] [77]	327
[0] [78]	359
[0] [79]	397
[0] [80]	443
[0] [81]	493
[0] [82]	549
[0] [83]	611
[0] [84]	680
[0] [85]	753
[0] [86]	832
[0] [87]	916
[0] [88]	1005
[0] [89]	1098
[0] [90]	1193
[0] [91]	1292
[0] [92]	1395
[0] [93]	1498
[0] [94]	1607
[0] [95]	1715
[0] [96]	1827
[0] [97]	1938
[0] [98]	2050
[0] [99]	2160

# 11. Accelerometro

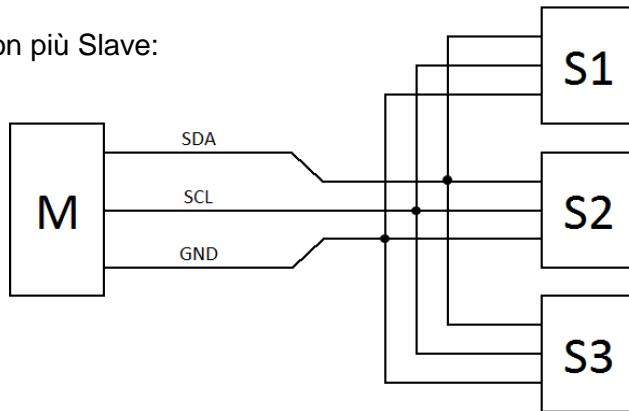
## 11.1 Interfaccia I<sup>2</sup>C

L'interfaccia del bus I<sup>2</sup>C (*Inter-Integrated Circuit*) si occupa delle comunicazioni tra il microcontrollore e il bus seriale I<sup>2</sup>C.

Il protocollo di comunicazione I<sup>2</sup>C è mappato sul bus APB1 e può essere schematizzato così:



Nel caso di un Master con più Slave:



Il segnale di clock è gestito sempre e solo dal Master, mentre la linea dati può essere gestita sia dal Master che dallo Slave, a seconda se si tratta di trasmissione o ricezione.

Un protocollo può essere:

- **Sincrono:** fornisce una transizione basso-alto ogni  $\mu$ s. Le periferiche si sincronizzano col fronte di salita
- **Asincrono:** il tempo tra una transizione e l'altra non è definita

L'I<sup>2</sup>C è un protocollo seriale sincrono.

Per poter usufruire dei canali SCL e SDA bisogna settare le porte PB6 e PB7 in modalità Alternate Function e selezionare AF4.

Port & Pin Name	AF0	AF1	AF2	AF3	AF4
PB0			TIM3_CH3	TSC_G3_IO2	TIM8_CH2N
PB1			TIM3_CH4	TSC_G3_IO3	TIM8_CH3N
PB2				TSC_G3_IO4	
PB3	JTDO-TRACE SWO	TIM2_CH2	TIM4_ETR	TSC_G5_IO1	TIM8_CH1N
PB4	NJTRST	TIM16_CH1	TIM3_CH1	TSC_G5_IO2	TIM8_CH2N
PB5		TIM16_BKIN	TIM3_CH2	TIM8_CH3N	I2C1_SMBA
PB6		TIM16_CH1N	TIM4_CH1	TSC_G5_IO3	I2C1_SCL
PB7		TIM17_CH1N	TIM4_CH2	TSC_G5_IO4	I2C1_SDA

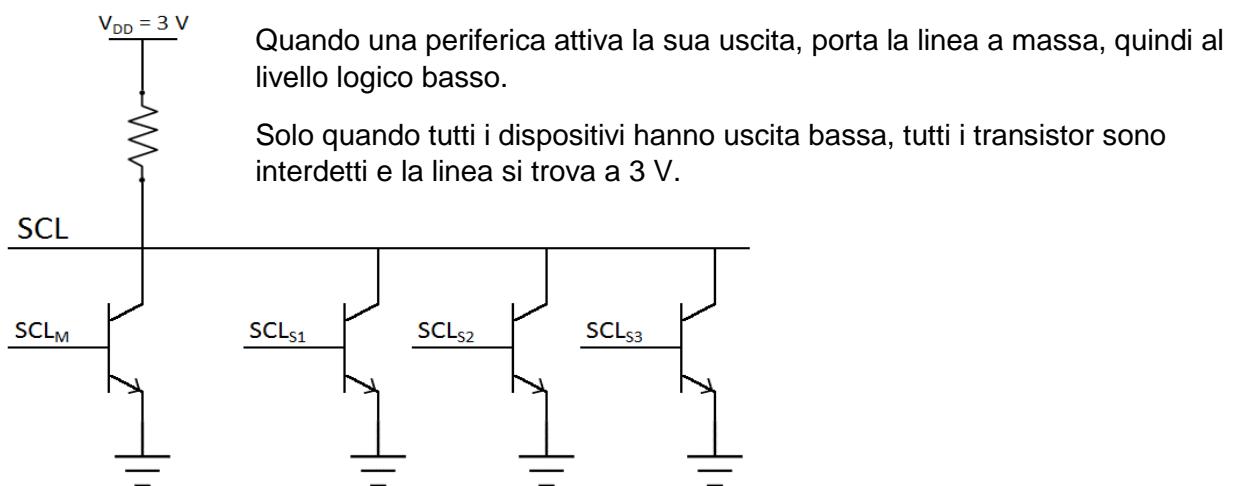
## Comunicazione e trasmissione dati:

La comunicazione avviene in questo modo: il Master trasmette l'indirizzo dello Slave con cui vuole comunicare e lo Slave che lo riconosce come il proprio indirizzo manda un segnale di ACK (*Acknowledge*). A questo punto avviene la vera e propria trasmissione o ricezione di dati tra il Master e lo Slave.

Il dispositivo che riceve i dati invia un segnale di ACK ogni 8 bit ricevuti.

Se il ricevente invia un segnale di NACK, invece:

- Se è lo Slave significa che non ha ricevuto il byte
- Se è il Master significa che vuole che questo sia l'ultimo byte da ricevere



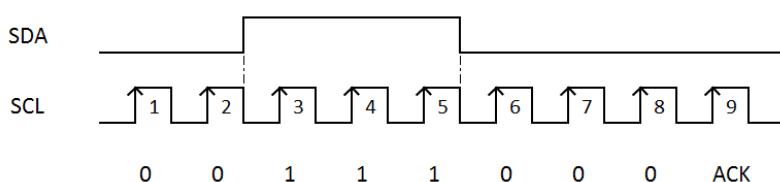
(Lo stesso disegno vale per la linea SDA)

Quindi se nessuno sta "parlando" SCL e SDA sono entrambe alte.

Nel caso della SCL, è il Master che controlla il clock, quindi gli Slave sono disattivati mentre il Master attiva e disattiva il transistor.

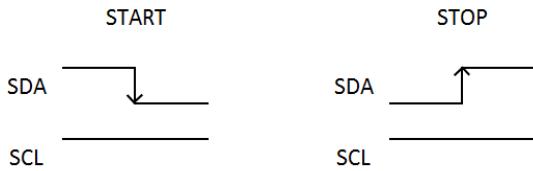
Nel caso della SDA, controlla la linea chi sta trasmettendo, chiunque altro è disattivato perché non deve interferire sulla linea portandola a '0' (basta che uno si attivi perché la linea sia bloccata a '0'). L'unica eccezione è che il ricevente, ogni byte ricevuto, manda il segnale ACK collegandosi alla linea e portandola conseguentemente a '0'.

ACK è dunque rappresentato dalla linea dati bassa (al nono bit). Del resto se la linea fosse alta vorrebbe dire che tutte le periferiche sono "staccate" e quindi nessuno starebbe inviando ACK.



Il dato varia quando il clock è basso ed è stabile quando il clock è alto.

Se invece il dato varia quando il clock è alto vuol dire che è stato segnalato un segnale di START o di STOP. Per avviare una comunicazione il Master invia un segnale di START, per interromperla invia invece un segnale di STOP.



Quando il Master trasmette allo Slave, il dato finisce nel registro TXDR (*Transmit Data Register*); quando riceve dallo Slave, il dato finisce nel registro RXDR (*Receive Data Register*).

Ci sono 2 flag associati a questi registri: TXE (*Transmit Register Empty*) e RXNE (*Receive Register Not Empty*), che segnalano rispettivamente che TXDR è vuoto e RXDR è pieno.

Per distinguere se si tratta di una trasmissione o una ricezione di dati si ricorre ad un bit R/W (1=R, 0=W).

Vediamo ora nello specifico come si comportano Master e Slave:

- **Trasmissione**

Master	START	ADD <sub>S</sub> + W		ADD <sub>R</sub>		DATA		DATA		STOP
Slave			ACK		ACK		ACK		ACK	

Dopo il segnale di inizio (START = 1), il Master deve indicare l'indirizzo dello Slave con cui vuole interagire (ADD<sub>S</sub>), che è costituito da 7 bit. L'ottavo bit del primo byte inviato è W.

Il bit di START si azzerà alla ricezione del primo ACK, ad indicare che il collegamento con lo Slave ha avuto successo.

Prima di scrivere dati in un registro dello Slave va inviato l'indirizzo di tale registro (ADD<sub>R</sub>). Ogni byte inviato dal Master (quindi ADD<sub>S</sub>, ADD<sub>R</sub>, W e i DATA) passa per il TXDR prima di arrivare allo Slave ed è seguito dall'ACK dello Slave.

Ogni volta che lo Slave riceve il dato leggendo TXDR, si alza il flag TXE.

- **Ricezione**

Master	START	ADD <sub>S</sub> + W		ADD <sub>R</sub>		START	ADD <sub>S</sub> + R		ACK	NACK	STOP
Slave			ACK		ACK		ACK	DATA		DATA	

Dapprima il Master si collega in scrittura per trasmettere l'indirizzo del registro che vuole leggere.

Successivamente, dopo un secondo START, si collega in lettura.

Ogni byte ricevuto è seguito da un ACK del Master, tranne l'ultimo che è seguito da un segnale di NACK.

I byte ricevuti finiscono nel RXDR, che viene poi letto dal Master.

Al termine di un'operazione vi è il segnale di STOP. Dopo il segnale di STOP si alza il flag STOPF, che deve essere abbassato attraverso il bit STOPCF.

## Timing:

Per di abilitare l'I<sup>2</sup>C bisogna settare il bit PE nel registro CR1, ma prima di farlo occorre configurare il clock del Master impostando:

- $SCL_H$
- $SCL_L$
- $SDA_{DEL}$
- $SCL_{DEL}$

$SCL_H$  e  $SCL_L$  quantificano il periodo del livello alto e basso del master clock.

$SDA_{DEL}$  e  $SCL_{DEL}$  rappresentano il ritardo del SDA e SCL.

A pag. 673 del Reference Manual c'è una tabella che mostra i vincoli che questi parametri devono rispettare, come ad esempio il minimo tempo per cui SCL deve rimanere basso o alto.

Per fortuna non abbiamo bisogno di ricorrere a tali misure: possiamo far riferimento a quella che la prof. Liccardo chiama "tabella ignorante" di pag. 684. Questa tabella ci mostra i vari parametri che dobbiamo inserire nel registro I2C\_TIMINGR.

Il clock del Master può operare in 3 modalità:

- Standard (fino a 100 KHz)
- Fast (fino a 400 KHz)
- Fast plus (fino a 1 MHz)

Di default il clock lavora a 100 KHz, quindi useremo sempre i valori della suddetta tabella ignorante.

Parameter	Standard mode		Fast Mode 400 kHz	Fast Mode Plus 500 kHz
	10 kHz	100 kHz		
PRESC	1	1	0	0
SCLL	0xC7	0x13	0x9	0x6
$t_{SCLL}$	200x250 ns = 50 $\mu$ s	20x250 ns = 5.0 $\mu$ s	10x125 ns = 1250 ns	7x125 ns = 875 ns
SCLH	0xC3	0xF	0x3	0x3
$t_{SCLH}$	196x250 ns = 49 $\mu$ s	16x250 ns = 4.0 $\mu$ s	4x125ns = 500ns	4x125 ns = 500 ns
$t_{SCL}^{(1)}$	~100 $\mu$ s <sup>(2)</sup>	~10 $\mu$ s <sup>(2)</sup>	~2500 ns <sup>(3)</sup>	~2000 ns <sup>(4)</sup>
SDADEL	0x2	0x2	0x1	0x1
$t_{SDADEL}$	2x250 ns = 500 ns	2x250 ns = 500 ns	1x125 ns = 125 ns	1x125 ns = 125 ns
SCLDEL	0x4	0x4	0x3	0x1
$t_{SCLDEL}$	5x250 ns = 1250 ns	5x250 ns = 1250 ns	4x125 ns = 500 ns	2x125 ns = 250 ns

## 11.2 Registri utili dell'I<sup>2</sup>C

I principali registri sono:

- **I2Cx\_CR1 (Control Register 1)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	PECEN	ALERT EN	SMBDEN	SMBHEN	GCEN	WUPEN	NOSTRETCH	SBC
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXDMA EN	TXDMA EN	Res.	ANF OFF		DNF		ERRIE	TCIE	STOP IE	NACK IE	ADDR IE	RXIE	TXIE	PE	
rw	rw		rw		rw		rw	rw	rw	rw	rw	rw	rw	rw	rw

- **PE (Peripheral Enable)**
  - 1: I<sup>2</sup>C abilitato

- **I2Cx\_CR2 (Control Register 2)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	PEC BYTE	AUTO END	RE LOAD								
					rs	rw	rw								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NACK	STOP	START	HEAD 10R	ADD10	RD_W RN										
rs	rs	rs	rw	rw	rw										

- **NBYTES (Number of bytes)**
- **STOP**
- **START**
- **RD\_WRN (Read / Not Write)**
  - 0: Write
  - 1: Read
- **SADD (Slave Address)** (solo i sette bit da 7 a 1)

- **I2Cx\_RXDR e I2C\_TXDR (Receive / Transmit Register)**

7	6	5	4	3	2	1	0
RXDATA[7:0]							
r							

7	6	5	4	3	2	1	0
TXDATA[7:0]							
rw							

- **I2Cx\_TIMINGR (Timing Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRESC[3:0]				Res	Res	Res	Res	SCLDEL[3:0]				SDADEL[3:0]			
rw				rw				rw				rw			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SCLH[7:0]				SCLL[7:0]				rw				rw			

- PRESC (*Timing Prescaler*)
- SCLDEL (*Clock Delay*)
- SDADEL (*Data Delay*)
- SCLH (*Clock High Period*)
- SCLL (*Clock Low Period*)

- **I2Cx\_ISR (Interrupt and Status Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	ADDCODE[6:0]							DIR
								r							r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BUSY	Res	ALERT	TIME OUT	PEC ERR	OVR	ARLO	BERR	TCR	TC	STOPF	NACKF	ADDR	RXNE	TXIS	TXE
r		r	r	r	r	r	r	r	r	r	r	r	r	r_w1	r_w1

- STOPF (*Stop Flag*)
  - 1: condizione di STOP rilevata
- RXNE (*Receive Data Register Not Empty*)
  - 1: RXDR è pieno
- TXE (*Transmit Data Register Empty*)
  - 1: TXDR è vuoto

- **I2Cx\_ICR (Interrupt Clear Register)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	ALERT CF	TIME OUTCF	PECCF	OVRCF	ARLO CF	BERR CF	Res	Res	STOP CF	NACK CF	ADDR CF	Res	Res	Res
w	w	w	w	w	w	w	w			w	w	w			

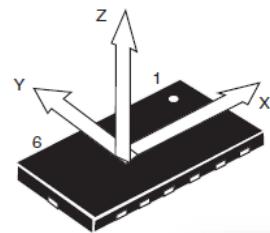
- STOPCF (*Stop Flag Clear*)
  - 1: Pulisce il flag STOPF

## 11.3 Accelerometro

Nel nostro caso il Master è rappresentato dal microcontrollore e lo Slave dall'accelerometro.

Per comunicare, il microcontrollore e l'accelerometro devono dunque utilizzare l'I<sup>2</sup>C.

L'indirizzo dell'accelerometro è 11001, cioè **0x19** (facile convertire in esadecimale se lo vediamo come 0001 1001).



SAD[7:1]
0011001

## 11.4 Registri utili dell'accelerometro

Gli indirizzi dei registri sono riportati sul manuale dell'accelerometro a pag. 22.

I registri che più ci interessano sono i 6 relativi alle coordinate x, y, z e il registro CTRL\_REG1\_A.

**Nota:** l'accelerometro permette di incrementare automaticamente l'indirizzo dei registri in caso di lettura. Per farlo si mette un '1' davanti all'indirizzo del primo registro da leggere.

I registri **OUT\_X\_L\_A** e **OUT\_X\_H\_A** contengono rispettivamente il byte meno significativo e quello più significativo dei 16 bit che indicano il valore dell'accelerazione sull'asse x. In realtà l'accelerazione è rappresentata solo dai primi **12 bit**, per cui una volta trovati letti i 16 bit dei registri bisogna effettuare una traslazione a destra di 4 bit. Il risultato è espresso in complemento a due, ma la conversione a intero con segno è automatica quando si va a creare la variabile in C. Analoga la situazione per y e z.

Name	Slave address	Type	Register
			Hex
Reserved (do not modify)	<a href="#">Table 14</a>		00 - 1F
CTRL_REG1_A	<a href="#">Table 14</a>	rw	20
CTRL_REG2_A	<a href="#">Table 14</a>	rw	21
CTRL_REG3_A	<a href="#">Table 14</a>	rw	22
CTRL_REG4_A	<a href="#">Table 14</a>	rw	23
CTRL_REG5_A	<a href="#">Table 14</a>	rw	24
CTRL_REG6_A	<a href="#">Table 14</a>	rw	25
REFERENCE_A	<a href="#">Table 14</a>	rw	26
STATUS_REG_A	<a href="#">Table 14</a>	r	27
OUT_X_L_A	<a href="#">Table 14</a>	r	28
OUT_X_H_A	<a href="#">Table 14</a>	r	29
OUT_Y_L_A	<a href="#">Table 14</a>	r	2A
OUT_Y_H_A	<a href="#">Table 14</a>	r	2B
OUT_Z_L_A	<a href="#">Table 14</a>	r	2C
OUT_Z_H_A	<a href="#">Table 14</a>	r	2D
EFO_CTRL_REG_A	<a href="#">Table 14</a>	rw	2E

Il registro **CTRL\_REG1\_A** (*Control Register 1*) è costituito dai seguenti bit:

ODR3	ODR2	ODR1	ODR0	LPen	Zen	Yen	Xen
------	------	------	------	------	-----	-----	-----

- ODR (*Data Rate Selection*)
  - 0000: spento
  - 0001-1001: acceso a varie frequenze (1, 10, 25, 50, 100, 200 ... Hz)
- LP<sub>EN</sub> (*Low Power Enable*)
- X<sub>EN</sub>, Y<sub>EN</sub>, Z<sub>EN</sub> (*x, y, z Axis Enable*)

## 11.5 Progetto 14: lettura accelerazione

Vogliamo abilitare l'accelerometro e leggere i valori delle accelerazioni sui 3 assi.

Le prime cose da fare sono:

- Abilitare il clock dell'I<sup>2</sup>C<sub>1</sub> e del GPIOB
- Impostare PB6 e PB7 in modalità Alternate Function
- Selezionare l'alternate function AF4 nei registri AFRL (il nome del registro è AFR[0])
- Configurare il registro TIMINGR con i valori della tabella "ignorante"
- Abilitare l'I<sup>2</sup>C settando PE in CR1

Ora dobbiamo configurare l'accelerometro, leggere i risultati ed elaborarli.

```
void abilitazione_periferiche() {
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;

void config_GPIO() {
    GPIOB->MODER |= GPIO_MODER_MODER6_1; //10: AF mode
    GPIOB->MODER |= GPIO_MODER_MODER7_1; //10: AF mode
    GPIOB->AFR[0] |= (4<<24); //0100: AF4
    GPIOB->AFR[0] |= (4<<28); //0100: AF4

void config_timing() {
    I2C1->TIMINGR |= (1<<28); //PRESC = 1
    I2C1->TIMINGR |= (0x4<<20); //SCLDEL = 0x4
    I2C1->TIMINGR |= (0x2<<16); //SDADEL = 0x2
    I2C1->TIMINGR |= (0xF<<8); //SCLH = 0xF
    I2C1->TIMINGR |= (0x13<<0); //SCLL = 0x13

I2C1->CR1 |= I2C_CR1_PE; //abilitazione I2C1
```

### Configurazione accelerometro

Ricorriamo al registro CTRL\_REG1\_A (che ha indirizzo 0x20) per abilitare l'acquisizione dei valori e abilitare tutti e 3 gli assi, inserendo:

- un qualsiasi numero diverso da 0 nei bit ODR (per esempio 0001, così da avere 1 Hz, cioè un valore al secondo)
- 1 in X<sub>EN</sub>, Y<sub>EN</sub>, Z<sub>EN</sub>

Quindi il contenuto del registro deve essere 0001 0111, cioè 0x17.

Il microcontrollore deve allora scrivere 0x17 nel registro di indirizzo 0x20 dello slave di indirizzo 0x19.

**Nota:** per sicurezza, prima di inserire l'indirizzo dello slave conviene azzerare il registro CR2

Prepariamoci alla trasmissione:

- Azzeriamo CR2
- Scriviamo '2' in NBYTES (dobbiamo trasmettere sia l'indirizzo del registro che il contenuto)
- Vogliamo trasmettere, quindi RD\_WRN = 0
- Inseriamo 0x19 nei 7 bit di SADD che vanno da 7 a 1 (quindi ignorando il bit 0)

Effettuiamo la trasmissione:

- START = 1
- Aspettiamo che START si abbassi
- Trasmettiamo l'indirizzo del registro (0x20) scrivendolo in TXDR
- Aspettiamo TXE
- Trasmettiamo il contenuto del registri (0x17) scrivendolo in TXDR

- Aspettiamo TXE
- STOP = 1
- Aspettiamo STOPF
- Puliamo STOPF col bit STOPCF

## Lettura registri degli assi

Leggiamo il contenuto dei 6 registri relativi agli assi.

Per semplicità attiviamo l'incremento automatico dell'indirizzo dei registri da leggere, così non dobbiamo ricorrere ad un for.

L'indirizzo del primo registro che ci interessa (OUT\_X\_L\_A) è 0x28, cioè 010 1000. Aggiungendo un 1 davanti si ottiene 1010 1000, cioè **0xA8**.

Se non volete usare l'incremento automatico fate un for in cui incrementate l'indirizzo manualmente (0x28 + i)

Prepariamoci al trasferimento della richiesta di lettura:

- Azzeriamo CR2
- Scriviamo '1' in NBYTES (solo l'indirizzo del registro)
- Vogliamo trasmettere, quindi RD\_WRN = 0
- Inseriamo 0x19 nei 7 bit di SADD che vanno da 7 a 1

Inviamo la richiesta di lettura:

- START = 1
- Aspettiamo che START si abbassi
- Trasmettiamo l'indirizzo del registro (0xA8) scrivendolo in TXDR
- Aspettiamo TXE
- Non effettuiamo lo STOP, così il prossimo START vale come RESTART

Prepariamoci alla ricezione:

- Azzeriamo CR2
- Scriviamo '6' in NBYTES (i 6 registri degli assi)
- Vogliamo ricevere, quindi RD\_WRN = 1
- Inseriamo 0x19 nei 7 bit di SADD che vanno da 7 a 1

Riceviamo:

- START = 1
- Aspettiamo che START si abbassi
- Aspettiamo RXNE
- Mettiamo il risultato nel primo elemento di un vettore di 6 elementi di 8 bit
- Iteriamo i precedenti 2 passaggi
- STOP = 1
- Aspettiamo STOPF
- Puliamo STOPF col bit STOPCF

## Elaborazione dei risultati

Dichiariamo 3 variabili intere di 16 bit: x, y, z.

```
int16_t x, y, z;
```

Ognuna di esse avrà al byte di sinistra il contenuto del corrispondente registro “alto” e al byte di destra quello del registro “basso”. Quindi trasliamo a sinistra di 8 bit il registro alto effettuando un **casting a 16 bit senza segno** (altrimenti la traslazione non avviene, perché traslerebbe al di fuori degli 8 bit, e senza segno perché è espresso in complemento a 2), poi sommiamo il registro basso. Infine bisogna effettuare un altro casting a intero a 16 bit **con segno** e solo a questo punto possiamo traslare a destra di 4, perché il risultato è contenuto nei primi 12 bit.

```
x = ( (int16_t) ((uint16_t)ris[1]<<8) + ris[0]) >> 4;  
y = ( (int16_t) ((uint16_t)ris[3]<<8) + ris[2]) >> 4;  
z = ( (int16_t) ((uint16_t)ris[5]<<8) + ris[4]) >> 4;
```

Il risultato è il valore delle componenti espressi in milli-g, quindi un valore vicino a 1000 rappresenta  $9.8 \text{ m/s}^2$  ( $1000\text{milli-g} = 1\text{g} = 9.8\text{m/s}^2$ ).

```

#include <stm32f30x.h>

void abilitazione_periferiche();
void config_GPIO();
void config_timing();
void setup_trasmissione(int);
void setup_ricezione(int);
void start();
void stop();
void trasmissione(int);
uint8_t ricezione();

int16_t x, y, z;           //globali per la live watch

void main(){
    uint8_t ris[6];

    abilitazione_periferiche();
    config_GPIO();
    config_timing();
    I2C1->CR1 |= I2C_CR1_PE;      //abilitazione I2C1

    //configurazione CTRL_REG1
    setup_trasmissione(2);
    start();
    trasmissione(0x20);
    trasmissione(0x17);
    stop();

    while(1){
        //lettura registri degli assi
        setup_trasmissione(1);
        start();
        trasmissione(0xA8);

        setup_ricezione(6);
        start();
        for(int i=0; i<6; i++)
            ris[i] = ricezione();
        stop();

        //elaborazione risultato
        x = ( (int16_t) ((uint16_t)ris[1]<<8) + ris[0]) >> 4;
        y = ( (int16_t) ((uint16_t)ris[3]<<8) + ris[2]) >> 4;
        z = ( (int16_t) ((uint16_t)ris[5]<<8) + ris[4]) >> 4;
    }
}

```

Expression	Value
x	0
y	0
z	1032

Expression	Value
x	-48
y	-44
z	-1000

Expression	Value
x	596
y	-520
z	616

– orizzontale  
 – orizzontale sottosopra  
 – posizione random

Lettura accelerazione

```

void abilitazione_periferiche(){
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN; //abilitazione GPIOB
    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN; //abilitazione I2C1
}

void config_GPIO(){
    GPIOB->MODER |= GPIO_MODER_MODER6_1; //10: AF mode
    GPIOB->MODER |= GPIO_MODER_MODER7_1; //10: AF mode
    GPIOB->AFR[0] |= (4<<24); //0100: AF4
    GPIOB->AFR[0] |= (4<<28); //0100: AF4
}

void config_timing(){
    I2C1->TIMINGR |= (1<<28); //PRESC = 1
    I2C1->TIMINGR |= (0x4<<20); //SCLDEL = 0x4
    I2C1->TIMINGR |= (0x2<<16); //SDADEL = 0x2
    I2C1->TIMINGR |= (0xF<<8); //SCLH = 0xF
    I2C1->TIMINGR |= (0x13<<0); //SCLL = 0x13
}

void setup_trasmissione(int n){
    I2C1->CR2 = 0;
    I2C1->CR2 |= (n<<16); //NBYTES
    I2C1->CR2 &= ~I2C_CR2_RD_WRN; //write
    I2C1->CR2 |= (0x19<<1); //accelerometro
}

void setup_ricezione(int n){
    I2C1->CR2 = 0;
    I2C1->CR2 |= (n<<16); //NBYTES
    I2C1->CR2 |= I2C_CR2_RD_WRN; //read
    I2C1->CR2 |= (0x19<<1); //accelerometro
}

void start(){
    I2C1->CR2 |= I2C_CR2_START; //START
    while((I2C1->CR2 & I2C_CR2_START) == I2C_CR2_START); //attesa START=0
}

void stop(){
    I2C1->CR2 |= I2C_CR2_STOP; //STOP
    while((I2C1->ISR & I2C_ISR_STOPF) != I2C_ISR_STOPF); //attesa STOPF
    I2C1->ICR |= I2C_ICR_STOPCF; //Clear STOPF
}

void trasmissione(int val){
    I2C1->TXDR = val;
    while((I2C1->ISR & I2C_ISR_TXE) != I2C_ISR_TXE); //attesa TXE
}

uint8_t ricezione(){
    while((I2C1->ISR & I2C_ISR_RXNE) != I2C_ISR_RXNE); //attesa RXNE
    return I2C1->RXDR;
}

```

E con questo siamo giunti al termine di *MAPI for Dummies*. Sperando di essere stato utile e comprensibile, vi auguro buona fortuna.