UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Misure per l'Automazione e la Produzione Industriale**

# *Programmazione di microcontrollori STM32: porte IO general purpose e timer*

Anno Accademico 2014-2015

Candidato:

**Antonio Russo**

**matr. N46001872**

[Dedica]

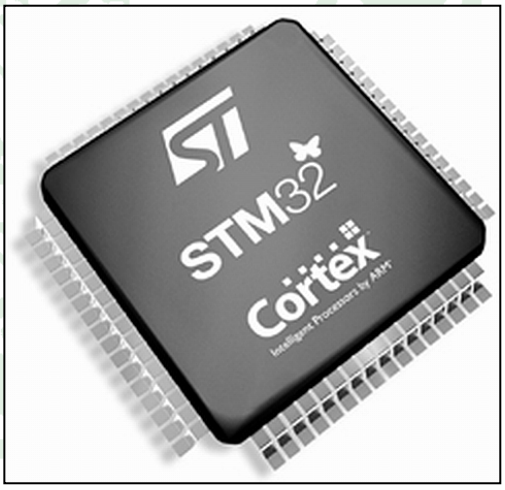
# Indice

# Introduzione

Nei primi anni ’70, quando nascevano i primi **microprocessori** progettati dalla Intel, nascevano anche i primi **microcontrollori**.

Lo Smithsonian Institution, istituto di istruzione e ricerca amministrato e finanziato dal governo degli Stati Uniti, afferma che il primo microcontrollore fu progettato da due ingegneri della Texas Instruments.

Ma che cos’è un microcontrollore? E’ un dispositivo elettronico integrato su singolo chip, nato come evoluzione alternativa al microprocessore. I primi esempi di microcontrollore erano semplicemente microprocessori con memoria incorporata, come RAM o ROM. Più tardi questi si sono evoluti in una vasta gamma di dispositivi utilizzati in una moltitudine di sistemi embedded, sistemi elettronici progettati appositamente per una determinata applicazione (special purpose), quali macchine, telefoni senza fili ed elettrodomestici.




Insomma la differenza tra un microprocessore ed un microcontrollore è che sicuramente il primo può essere utilizzato per applicazioni a scopo generale più ampie, mentre il secondo, avendo CPU, memorie e pin I/O, tutte su un unico chip, nasce per applicazioni più specifiche.

Infatti il microcontrollore è progettato per interagire direttamente con il mondo esterno tramite un programma residente nella propria memoria interna e mediante l'uso di pin specializzati o configurabili dal programmatore.

Nel corso degli ultimi sei o sette anni, una delle maggiori tendenze nella progettazione di microcontrollori general purpose, ossia ad uso generale, è stata l'adozione dell'architettura ARM7 e ARM9 come CPU.

L'architettura ARM, indica una famiglia di microprocessori a 32-bit, sviluppata da ARM Holdings. Oggi ci sono circa 240 microcontrollori basati su architettura ARM disponibili da una vasta gamma di produttori, tra i quali la STMicroelectronics.

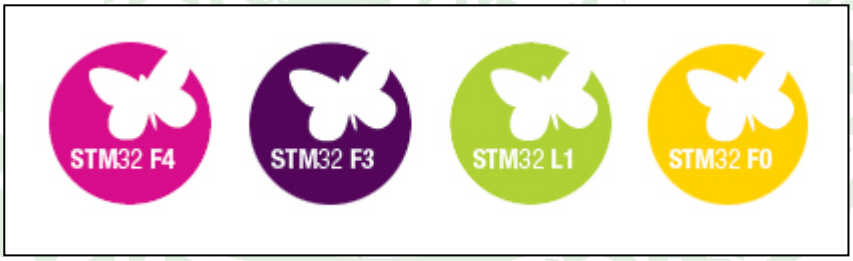
L'**STMicroelectronics** è un'azienda franco-italiana con sede a Ginevra (Svizzera), che produce componenti elettronici a semiconduttore, come i microcontrollori. Tra le famiglie di microcontrollori a 32-bit prodotti dall'azienda negli ultimi anni, abbiamo l'STM32 che è il loro primo microcontrollore basato sul nuovo core ARM Cortex-M3 che garantisce nuovi standard per quanto riguarda costi e performance, oltre ad essere in grado di operare a basso consumo.

Lo scopo di questa tesi sarà per l'appunto quella di avere un primo approccio con un microcontrollore, nello specifico l'**STM32**, capirne i vantaggi e la semplicità di utilizzo. In particolare useremo la board **F3 Discovery** di cui ne esplicheremo le funzionalità e potenzialità. Inoltre andremo ad analizzare nel dettaglio le sue **porte IO General Purpose** e i **Timer**, riportando esempi di configurazione ed utilizzo pratico.

# Capitolo 1: Microcontrollore STM32

L'STM32 della STMicroelectronics è il primo microcontrollore dell'azienda franco-italiana basato sul nuovo core **ARM Cortex-M3** che garantisce nuovi standard per quanto riguarda costi e performance, oltre ad essere in grado di operare a basso consumo. Attualmente l'STM32 ha più di 75 varianti con altre già annunciate. Queste sono divise in 4 gruppi:

**Performance Line** che opera a velocità di clock CPU fino a 72MHz, **Access Line** che invece opera fino a 36MHz, **USB Access Line** che aggiunge una periferica USB ed opera ad una velocità di clock CPU fino a 48MHz. L'ultimo gruppo, chiamato **Connectivity Line**, aggiunge periferiche di comunicazione avanzate che includono controller ETHERNET MAC e USB Host/OTG. Tutte le varianti offrono FLASH ROM di grandezza fino a 512K e 64K SRAM.



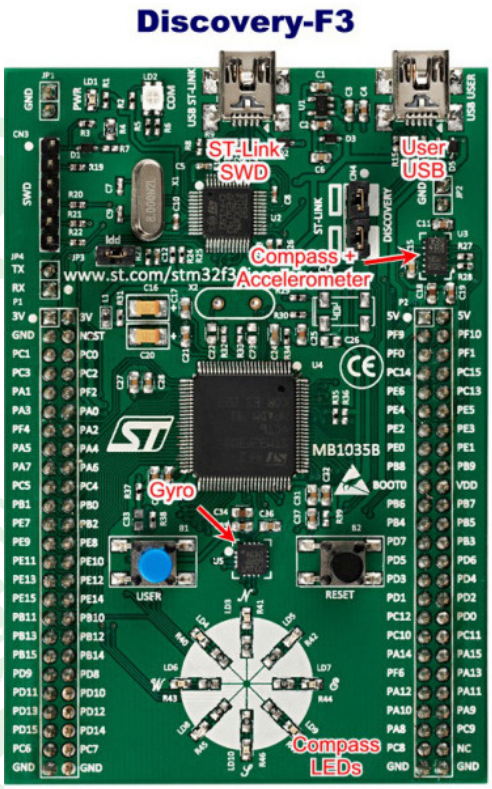

## 1.1 Board e Microcontrollore

E' importante fare innanzitutto una distinzione tra la board e il microcontrollore. Si potrebbe fare l'errore di pensare al microcontrollore come se stessimo parlando dell'intera board. Invece il microcontrollore è soltanto una parte dei circuiti integrati che costituiscono una board.

### 1.1.1 Board STM32F3Discovery

Come accennato in precedenza, la board oggetto di questa tesi è la **F3 Discovery**. Le caratteristiche chiave di questa board sono le seguenti:

- Microcontrollore **STM32F303VCT6** con 256 KB Flash, 48 KB RAM
- Fonte di alimentazione: attraverso il bus USB o ingresso esterno di alimentazione 3 V o 5 V
- L3GD20, ST MEMS sensore di movimento, giroscopio digitale a 3-assi
- Dieci LED
- LED1 (rosso) per lo stato di accensione da 3.3 V
- LED2 (rosso/verde) per comunicazioni USB
- Otto LED, LED3/10 (rosso), LED4/9 (blu), LED5/8 (arancione) and LED6/7 (verde)
- Due Bottoni (utente e reset)



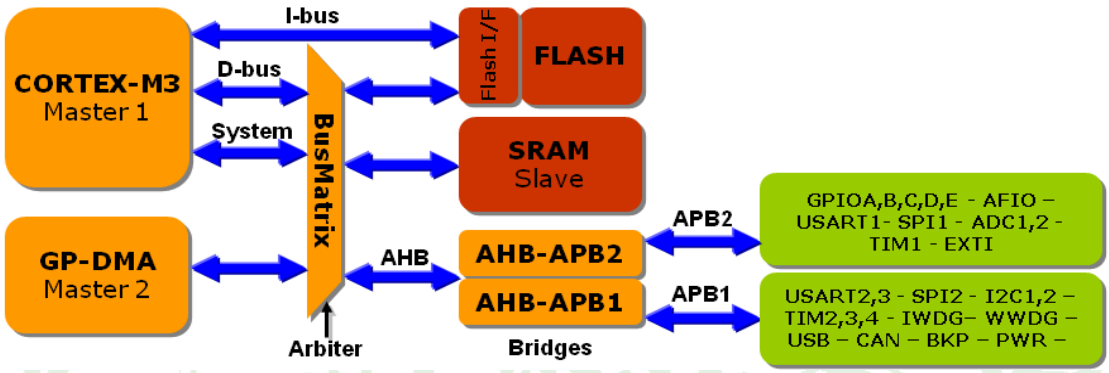

## 1.2 Complessità della Board

A prima vista il set di periferiche della **F3 Discovery** si presenta come quello di un tipico microcontrollore dalle dimensioni ridotte, con **periferiche** come **Dual ADC**, **General Purpose Timers**, **I2C**, **SPI**, **CAN**, **USB** ed un **Clock Real-Time**. Tuttavia, a dispetto di quanto possa sembrare, ognuna di queste periferiche è molto ricca di funzionalità. Un'ulteriore sorpresa per un microcontrollore così piccolo è data dal fatto che l'STM32 ha inclusa al suo interno un'unità **DMA** (Direct Memory Access) fino a 12 canali.

Un canale DMA è una connessione diretta tra dispositivi periferici e memoria che permette di trasferire dati da e per la memoria senza passare attraverso il processore.

## 1.3 System Architecture del Microcontrollore

**L'STM32** è composto da un core **Cortex** che è connesso alla memoria FLASH attraverso un bus Instruction (**I-bus**) dedicato. Lo stesso Cortex è collegato attraverso un bus Data (**D-bus**) e un bus di Sistema ad una matrice di bus, che attraverso l'**arbiter**, un meccanismo di arbitraggio, che permette di minimizzare la negoziazione tra le richieste di accesso ai dati in memoria tra la **CPU** e i canali **DMA**.




**(Figura 1)**

In pratica per il corretto funzionamento, un solo dispositivo fra quelli collegati allo stesso bus, può agire come master. Se esistono due o più masters, è necessario un meccanismo di arbitraggio. In questo caso sia il core Cortex che l'unità DMA possono essere entrambi bus master. La memoria **SRAM** interna è collegata direttamente al bus della matrice **AHB** (advanced high-performance bus), come lo è anche l'unità DMA. Le periferiche sono localizzate su due APB (Advanced Peripheral Busses). Ciascuno dei bus **APB** è connesso al bus della matrice AHB. Il bus della matrice AHB ha la stessa velocità di clock del core Cortex. Tuttavia, i bus AHB hanno prescalers separati e quindi possono essere usati a velocità di clock più basse per conservare energia.

E' importante sottolineare che l'**APB2** può funzionare fino a 72MHz mentre l'**APB1** è limitato a 36MHz.

Come detto in precedenza sia il core Cortex che l'unità DMA possono essere bus master ed essere gestiti attraverso meccanismo di arbitraggio, nell'eventualità che entrambi provino ad accedere contemporaneamente alla SRAM, all'APB1 o APB2. Tuttavia il meccanismo di arbitraggio garantirà i 2/3 del tempo di accesso ai dati all'unità DMA mentre il restante 1/3 alla CPU.

## 1.4 Sicurezza

Oltre a richiedere maggiore potenza e periferiche sempre più sofisticate, spesso si richiedono situazioni di operatività in ambienti critici per la sicurezza. Proprio per questo, l'STM32 ha una serie di caratteristiche hardware a supporto di tali situazioni che aiutano a preservare l'integrità e l'operatività nelle diverse applicazioni di utilizzo critiche. Tra queste abbiamo un rilevatore di bassa tensione, un sistema di sicurezza del clock e due **watchdog** separati, dispositivi che per l'appunto proteggono il sistema da specifici problemi software o hardware, facendo da "**guardia**" al corretto funzionamento dell'applicazione.

## 1.5 Sviluppo Software

L'STMicroelectronics fornisce diverse librerie per lo sviluppo software. Tra queste abbiamo una libreria con i driver delle periferiche e una libreria per lo sviluppo con la porta USB, tutte **librerie C**. Il Cortex è fornito di un nuovo sistema di **debug** chiamato **CoreSight**. L'accesso a suddetto sistema avviene attraverso la porta di accesso debug che supporta sia la connessione standard **JTAG** oppure un'interfaccia **seriale** a 2 pin.

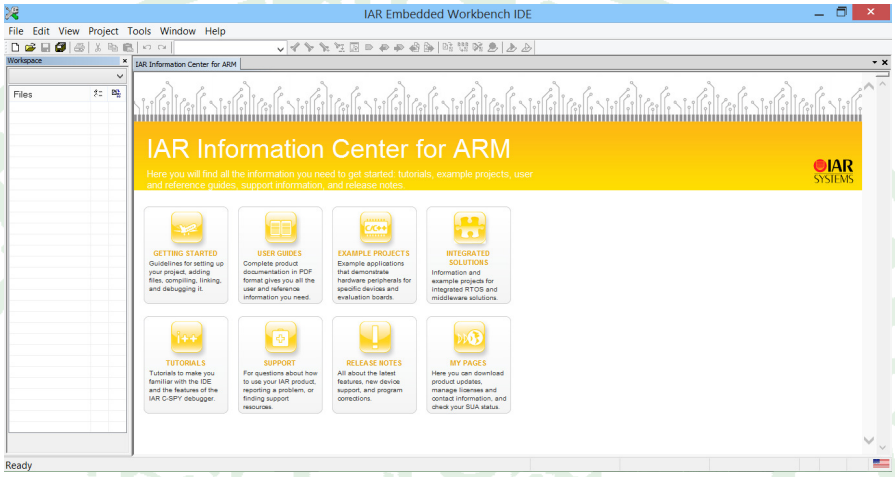
Grazie a questo sistema possiamo avere tantissime informazioni di debug che possono essere usate per il testing software.

### 1.5.1 Ambiente di Sviluppo IAR

Il software che utilizzeremo per la programmazione della board **sarà IAR Embedded Workbench**. Si tratta di un tool di sviluppo di proprietà della **IAR Systems**, una software company svedese che è stata fondata nel 1983. Esso ci permetterà di programmare in linguaggio C.

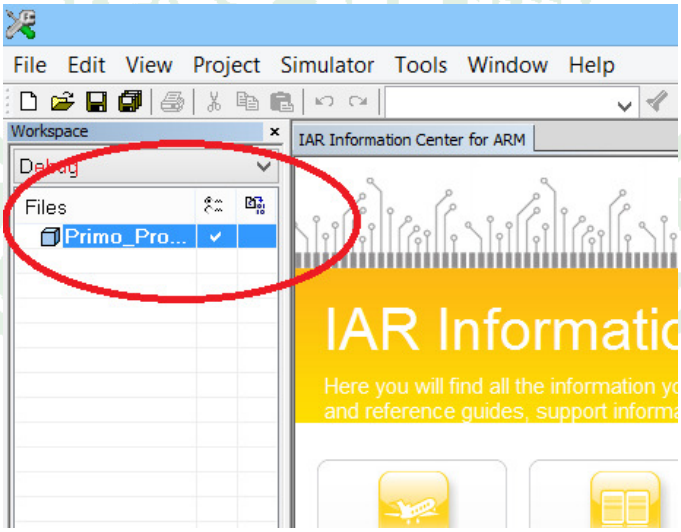
### 1.5.2 Realizzazione di un Nuovo Progetto

Per poter essere operativi ed utilizzare la nostra board con il software IAR Workbench, sarà necessario configurare alcuni parametri durante la creazione di un nuovo progetto. Diamo uno sguardo all'interfaccia del software e alle schermate di configurazione. Aperto l'ambiente di sviluppo IAR, si presenterà davanti ai nostri occhi la seguente interfaccia utente:



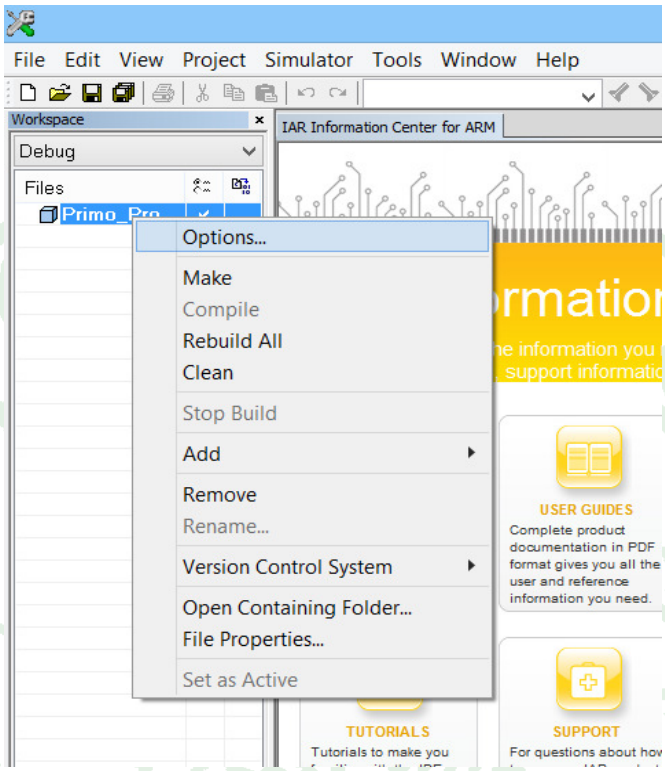

**(Figura 2)**

Una volta creato un nuovo workspace e salvato il nostro nuovo progetto, possiamo procedere alla configurazione delle opzioni così da renderlo compatibile con la nostra board in fase di caricamento e debug del programma.



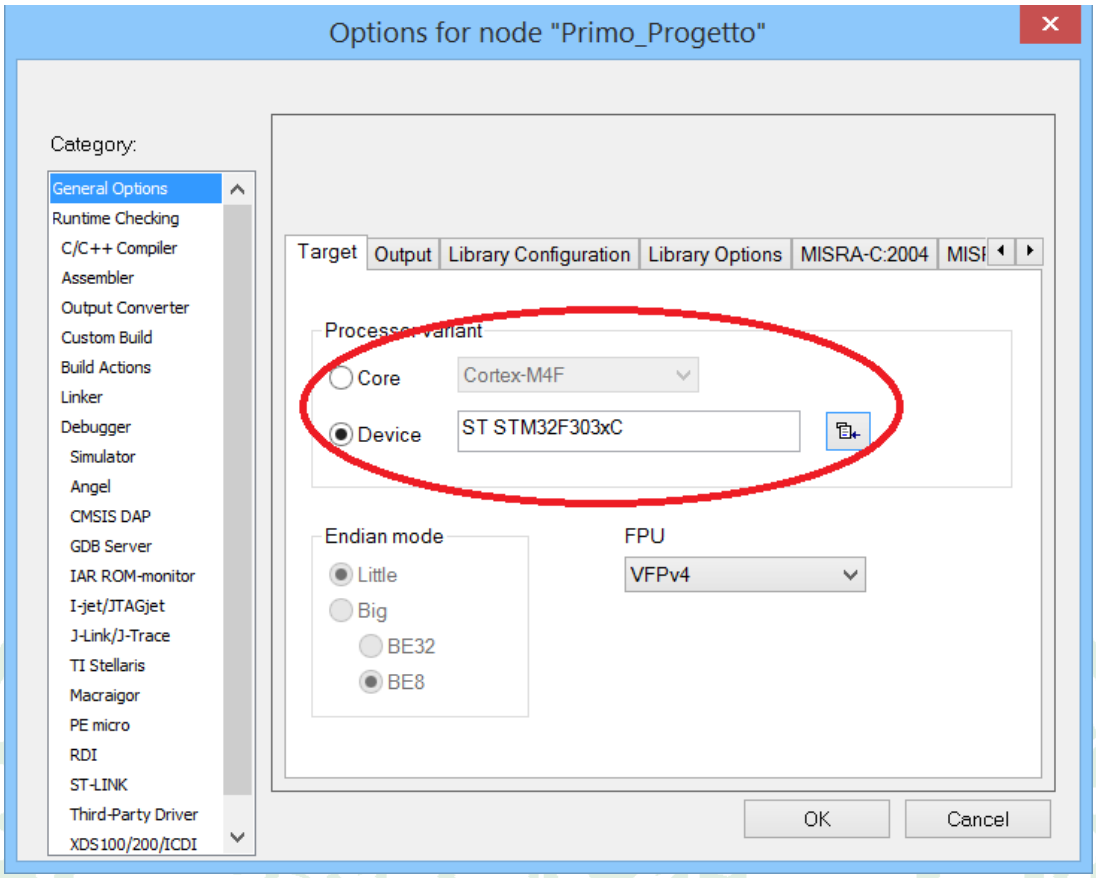

**(Figura 3)**

Ci basterà cliccare con il tasto destro sul nome del progetto all'interno del workspace e selezionare l'opzione **Options**.




**(Figura 4)**

Entrati nella schermata delle opzioni, dovremmo innanzitutto selezionare come **device** la voce **STM32F303xC** dall'elenco, che è proprio la nostra board.




**(Figura 5)**

Fatto ciò ci basterà configurare il **Preprocessor** che troviamo nella categoria **C/C++ Compiler**, in modo da selezionare le cartelli a cui il compilatore deve fare riferimento.

Infine nella categoria **Debugger** ci basterà spuntare la voce "**Use flash loader(s)**" e nella categoria **ST-LINK** selezionare l'interfaccia di comunicazione **SWD**.

Configurato il nostro progetto possiamo partire con la creazione del nostro **main** e la scrittura del codice del nostro programma.

Come progetto andremo a realizzare un programma che permetta l'accensione dei **led** sulla board, ma prima dobbiamo parlare delle **porte GPIO**, argomento che affronteremo nel prossimo capitolo.

# Capitolo 2: Porte IO General Purpose

GPIO è l'acronimo di **General Purpose I/O** e denota il concetto secondo il quale un pin può essere utilizzato per diversi scopi (**input, output**).

Per convenienza possiamo dividere le periferiche utente dell'STM32 in due gruppi:

**Periferiche General Purpose** e **Periferiche di Comunicazione**.

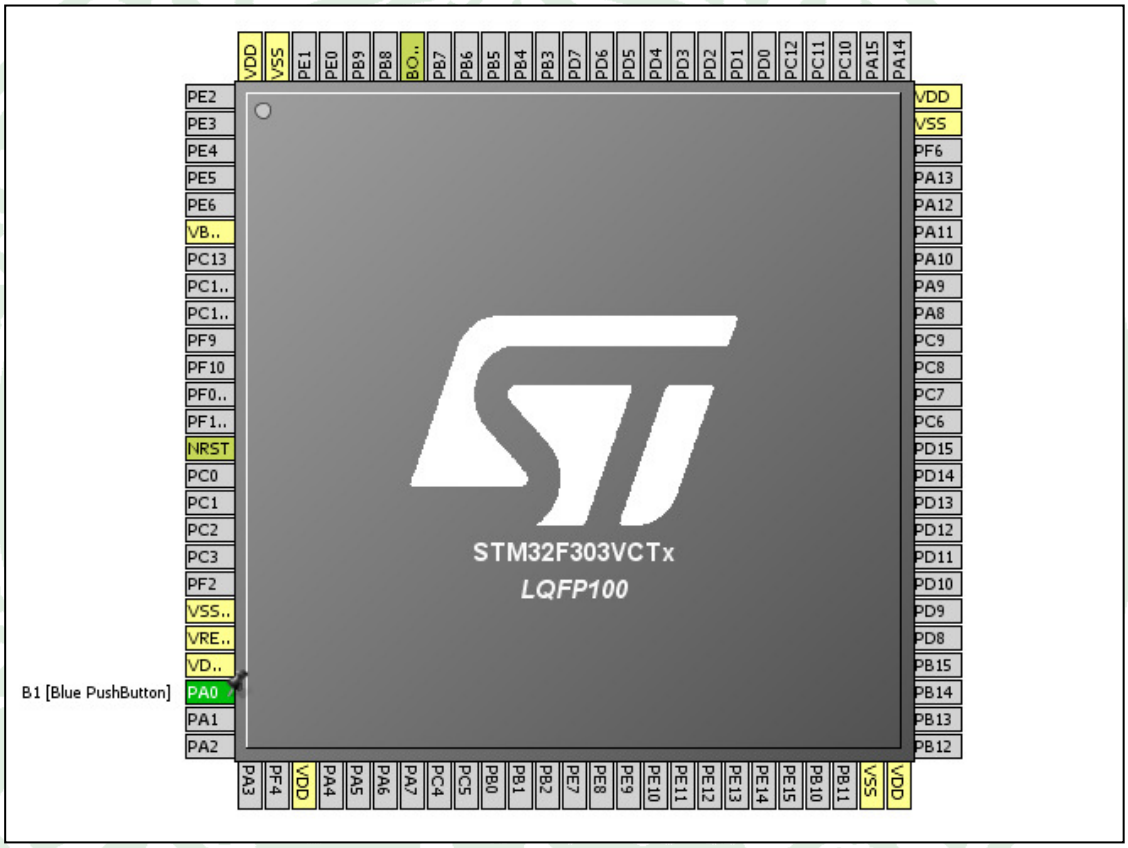
Le General Purpose sono costituite da alcune delle seguenti periferiche: porte IO general purpose, controller delle interrupt, convertitore ADC, unità timer general purpose ed avanzate e convertitore DAC. Nello specifico ci occuperemo delle porte I/O (Input/Output) General Purpose.

L'STM32 è composto da **96 pin IO** (Input/Output), quindi **bidirezionali**, che sono disposti su **6 porte**, ognuna avente **16 linee IO**. Queste porte sono etichettate a partire dalla lettera **A** alla lettera **F**.

Ciascun pin digitale può essere configurato sia come **GPIO** o come una **alternate function**. In seguito capiremo la differenza tra le due configurazioni. Inoltre tutti i pin possono essere configurati simultaneamente con una delle linee di interrupt esterne.

La funzionalità e lo stato di ogni pin è controllato da un numero di registri GPIO di cui parleremo nel dettaglio più avanti.

Per avere un'idea più chiara di **porte** e **pin** associati a suddette porte del microcontrollore, utilizziamo il software **STM32 Cube**, fornito dalla STMicroelectronics, che ci permette di generare il codice C di inizializzazione per la nostra board attraverso un interfaccia grafica ed intuitiva, e tra le varie funzionalità, nella sezione **pinout**, ci mostra il microcontrollore con i suoi pin, permettendoci di configurarli a nostro piacimento.




**(Figura 6)**

Ad esempio il pin **PA0** fa riferimento alla porta **GPIOA** ed è collegato al pulsante utente di colore blu presente sulla board. In **Figura 6**, questo risulta configurato come dispositivo di Input.

## 2.1 Alternate Functions

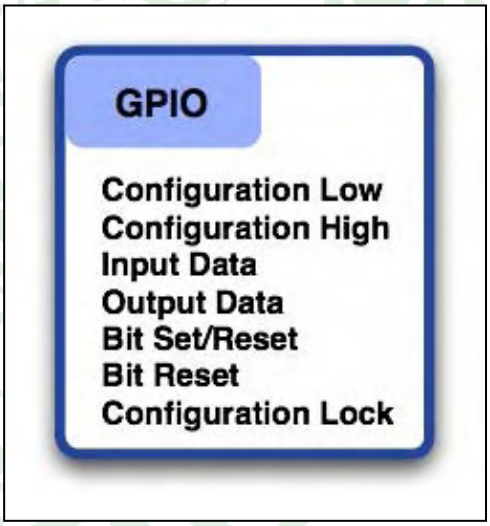
Il meccanismo delle alternate functions consente di connettere le periferiche con i pin di I/O del microcontrollore al fine di avere più pin logici mappati su di un numero minore di pin fisici.

Tale meccanismo risulta d'obbligo per cercare di rendere utilizzabili tutte le periferiche ottimizzando il numero di pin; in aggiunta, sempre per flessibilità, ad ogni periferica è associata una **AF** (Alternate Function) che può essere mappata su diversi pin di I/O.

In totale sono disponibili 16 AF (AF0...AF15) di cui solo AF1...AF13 sono utilizzabili dalle periferiche e l'associazione AF-Pin I/O avviene per mezzo di multiplexer pilotati da particolari registri di selezione associati a ciascuna porta che detiene il pin.

In aggiunta si ricordi che al reset del microcontrollore (e quindi di default) a ciascun pin è collegata la AF0 (System's AF).

## 2.2 GPIO Functional Description




Attraverso il "**Reference Manual"** fornito dalla STMicroelectronics possiamo analizzare in che modo è possibile configurare ognuna delle 6 porte GPIO in modo da avere un determinato tipo di funzionamento. Ciascun pin di una porta GPIO può essere configurato individualmente come input o output ed avere configurazioni del driver differenti. Tutto ciò è possibile grazie ai registri di configurazione. Ad ogni porta **GPIOx** ($x$= $A,B,C,D,E,F$) è associato un **set di registri** di configurazione che andremo a vedere nello specifico più avanti, quando mostreremo un esempio pratico di settaggio.

## 2.4 Esempio Pratico di Accensione dei Led sulla Board

Dopo aver capito come si realizza un progetto vuoto su IAR nel primo capitolo e dopo aver approfondito l'argomento delle porte GPIO in questo secondo capitolo, il passo successivo sarà quello di implementare un progetto che permetta l'accensione di uno o più led sulla board. L'intento è quello di mostrare un approccio di "**basso livello**", ovvero mostrare come il microcontrollore "**ragiona**" a livello di registri e linee di trasmissione, per poi abbandonare questo tipo di ragionamento dando maggiore spazio alle sottofunzioni già implementate in ambiente ARM.

Importanti, ai fini pratici della realizzazione, sono stati il "**Reference Manual**" e l' "**User Manual**" della board in uso.

Riportiamo la seguente tabella in **Figura 7**, per la sua importanza all'interno dell' User Manual piché associa a ciascun pin presente sulla board un particolare "**device**" ad esso collegato. Ad esso sono anche associate delle "alternate functions", la cui funzionalità è stata esplicata in precedenza.

**Table 6. STM32F303VCT6 MCU pin description versus board function**

| MCU pin | | | Board function | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Main function | Alternate functions | LQFP100 pin num. | LSM303DLHC | L3GD20 | Pushbutton | LED | SWD | USB | OSC | Free I/O | Power supply | CN3 | P1 | P2 |
| BOOT0 | | 94 | | | | | | | | | | | | 19 |
| NRST | | 14 | | | RESET | | NRST | | | | | 5 | 4 | |
| PA0 | TIM2_CH1_ETR, G1_IO1, USART2_CTS, COMP1_OUT, TIM8_BKIN, TM8_ETR | 23 | | | USER | | | | | | | | 12 | |
| PA1 | TIM2_CH2, G1_IO2, USART2_RTS, TIM15_CH1N | 24 | | | | | | | | | | | 9 | |
| PA2 | TIM2_CH3, G1_IO3, USART2_TX, COMP2_OUT, TIM15_CH1, AOP1_OUT | 25 | | | | | | | | | | | 14 | |
| PA3 | TIM2_CH4, G1_IO4, USART2_RX, TIM15_CH2 | 26 | | | | | | | | | | | 11 | |
| PA4 | TIM3_CH2, G2_IO1, SPI1_NSS, SPI3_NSS/I2S3_WS, USART2_CK | 29 | | | | | | | | | | | 16 | |
| PA5 | TIM2_CH1_ETR, G2_IO2, SPI1_SCK | 30 | | SCL/SPC | | | | | | | | | 15 | |

**(Figura 7)**

Ciò che a noi servirà nello specifico è questa parte della tabella dove possiamo vedere i pin che fanno riferimento ai led sulla board:

**Table 6. STM32F303VCT6 MCU pin description versus board function (continued)**

| MCU pin | | | Board function | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Main function | Alternate functions | LQFP100 pin num. | LSM303DLHC | L3GD20 | Pushbutton | LED | SWD | USB | OSC | Free I/O | Power supply | CN3 | P1 | P2 |
| PE8 | TIM1_CH1N | 39 | | | | LD4/ BLUE | | | | | | | 26 | |
| PE9 | TIM1_CH1 | 40 | | | | LD3/ RED | | | | | | | 25 | |
| PE10 | TIM1_CH2N | 41 | | | | LD5/ ORANGE | | | | | | | 28 | |
| PE11 | TIM1_CH2 | 42 | | | | LD7/ GREEN | | | | | | | 27 | |
| PE12 | TIM1_CH3N | 43 | | | | LD9/ BLUE | | | | | | | 30 | |
| PE13 | TIM1_CH3 | 44 | | | | LD10/ RED | | | | | | | 29 | |
| PE14 | TIM1_CH4_BKIN2 | 45 | | | | LD8/ ORANGE | | | | | | | 32 | |
| PE15 | TIM1_BKIN, USART3_RX | 46 | | | | LD6/ GREEN | | | | | | | 31 | |

**(Figura 8)**

Il manuale di cui però si farà maggiormente uso durante la realizzazione di questo progetto, sarà il Reference Manual, poiché schematizza ed analizza per ciascun registro qualsiasi possibile settaggio di ciascun dispositivo presente sulla board.

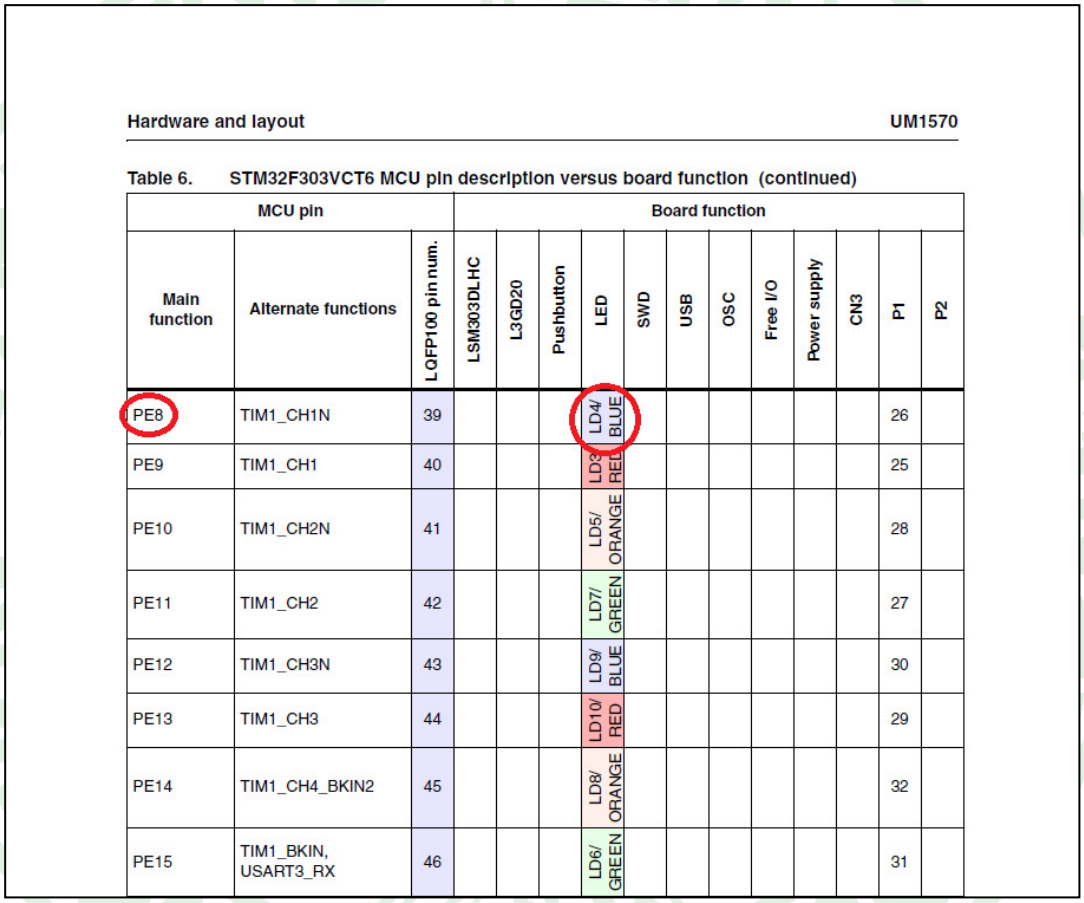
### 2.4.1 Approccio a Basso Livello

Dopo questa breve introduzione procediamo quindi nel mostrare come si "ragiona" quando si programmano microcontrollori. La board che utilizziamo è "**memory mapped**", ciò significa che a ciascun dispositivo è associato un indirizzo in memoria. Quindi possiamo "**puntare**" ad un particolare indirizzo ed andare a modificare il "**registro puntato**" in maniera tale da

svolgere la funzione richiesta.

La prima cosa che dobbiamo fare per **abilitare** un dispositivo sulla board è **attivare il clock** di quel particolare dispotivo.

Questa è una cosa **obbligatoria** e che rende necessario capire come abilitarlo, ovviamente dopo aver individuato il dispositivo da attivare. Ad esempio se volessimo scegliere di accendere il **led blu**, ci basterà prendere l'User Manual ed arrivare alla parte della tabella citata precedentemente in **Figura 8**.



Hardware and layout UM1570

Table 6. STM32F303VCT6 MCU pin description versus board function (continued)

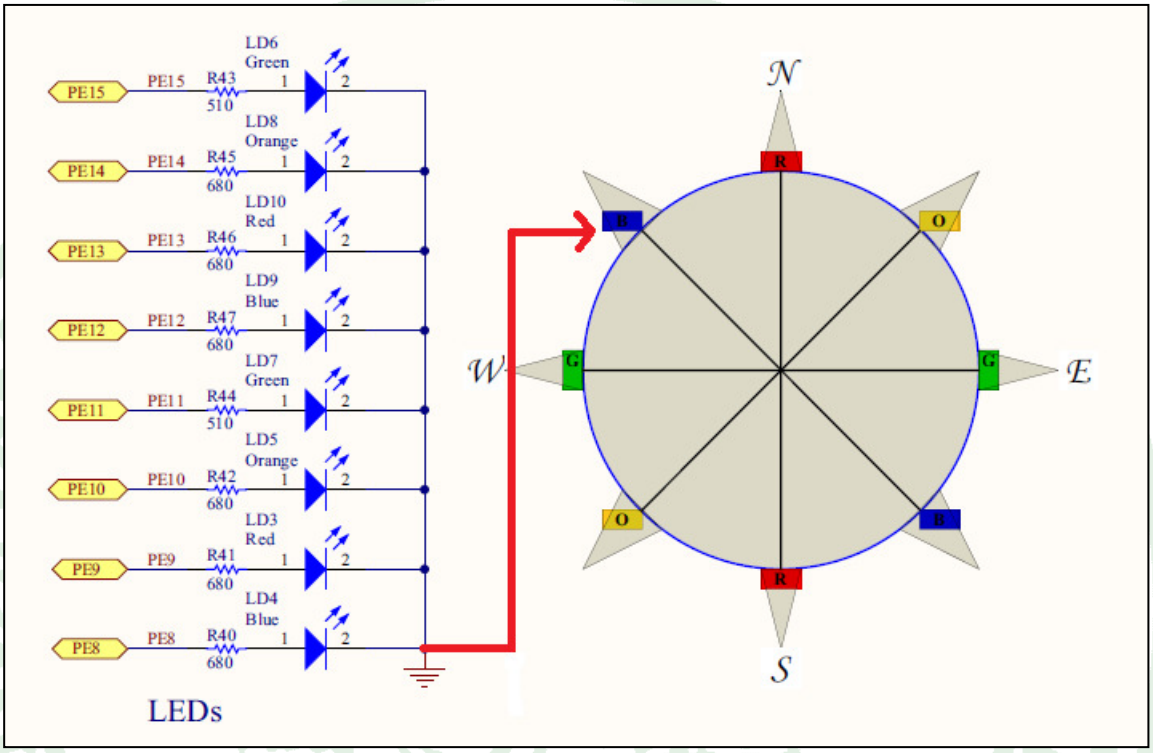
| MCU pin | | | Board function | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Main function | Alternate functions | LQFP100 pin num. | LSM303DLHC | L3GD20 | Pushbutton | LED | SWD | USB | OSC | Free I/O | Power supply | CN3 | P1 | P2 |
| PE8 | TIM1_CH1N | 39 | | | | LD4/ BLUE | | | | | | | 26 | |
| PE9 | TIM1_CH1 | 40 | | | | LD3/ RED | | | | | | | 25 | |
| PE10 | TIM1_CH2N | 41 | | | | LD5/ ORANGE | | | | | | | 28 | |
| PE11 | TIM1_CH2 | 42 | | | | LD7/ GREEN | | | | | | | 27 | |
| PE12 | TIM1_CH3N | 43 | | | | LD9/ BLUE | | | | | | | 30 | |
| PE13 | TIM1_CH3 | 44 | | | | LD10/ RED | | | | | | | 29 | |
| PE14 | TIM1_CH4_BKIN2 | 45 | | | | LD8/ ORANGE | | | | | | | 32 | |
| PE15 | TIM1_BKIN, USART3_RX | 46 | | | | LD6/ GREEN | | | | | | | 31 | |

**(Figura 9)**

Evidenziato in rosso troviamo il LD4/BLUE che risulta essere associato al pin PE8. Di conseguenza gli altri led saranno associati ai restanti pin come mostrato in Figura 9. Queste sigle saranno importanti per capire a quali indirizzi dovremmo puntare per il registro di interesse, in particolare quali bit devo modificare nel registro per espletare la funzione richiesta.

## 2.4.2 Abilitazione del Clock

Prima di vedere come abilitare il clock di quello specifico led o di più led osserviamo lo schematico dall'User Manual:



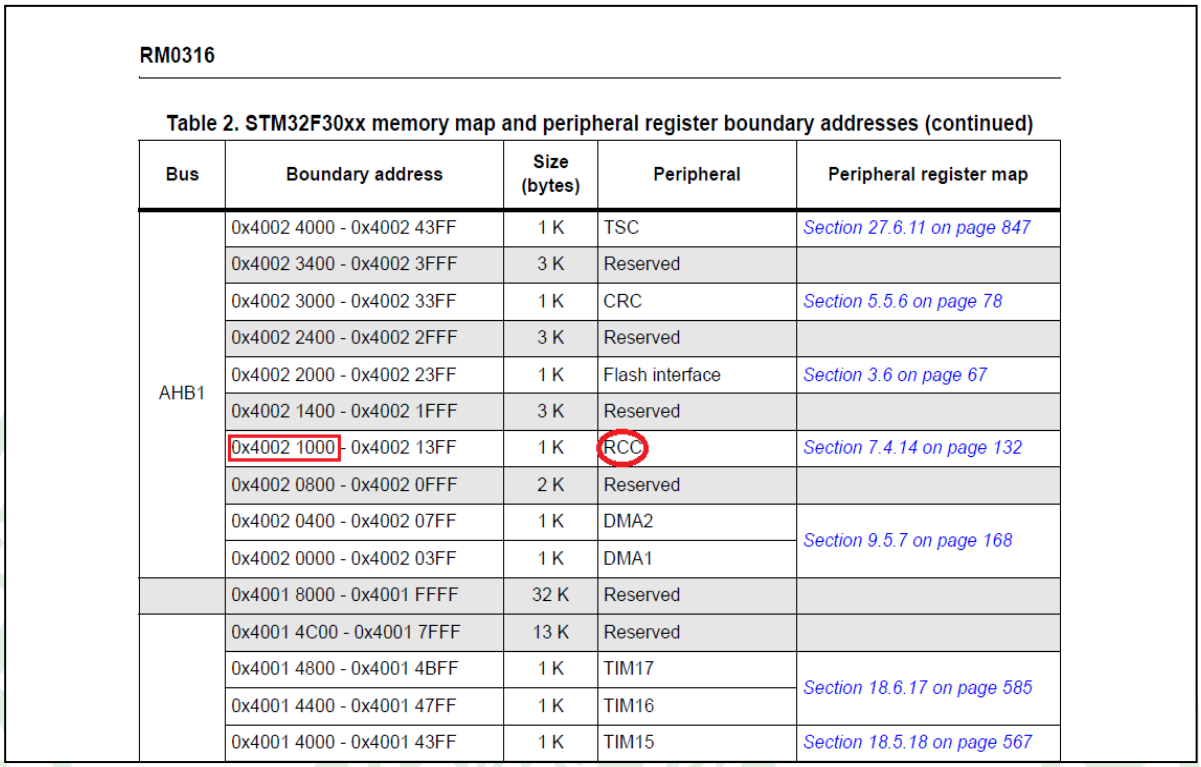

**(Figura 10)**

Questo ci aiuta subito a capire cosa stiamo facendo, ovvero stiamo andando ad abilitare la linea PE8 che si collega al led blu. Un collegamento in rosso evidenzia schematicamente il collegamento esistente tra la linea PE8 e il led blu. Lo stesso vale per gli altri led.

Proseguiamo quindi con l'abilitare il clock dei diversi led, per poterlo fare dobbiamo individuare l'**RCC**, ovvero il Reset and Clock Control, che di fatto rappresenta una periferica interna alla board che gestisce il reset e il clock di altre periferiche.

Possiamo trovare l'indirizzo in memoria che punta all'RCC attraverso una tabella nel Reference Manual.

In questa tabella troveremo tutti gli indirizzi di base di ciascun dispositivo presente sulla board. Ciò che ci interessa a noi è questa parte qui della tabella:



RM0316

Table 2. STM32F30xx memory map and peripheral register boundary addresses (continued)

| Bus | Boundary address | Size (bytes) | Peripheral | Peripheral register map |
|---|---|---|---|---|
| AHB1 | 0x4002 4000 - 0x4002 43FF | 1 K | TSC | Section 27.6.11 on page 847 |
| | 0x4002 3400 - 0x4002 3FFF | 3 K | Reserved | |
| | 0x4002 3000 - 0x4002 33FF | 1 K | CRC | Section 5.5.6 on page 78 |
| | 0x4002 2400 - 0x4002 2FFF | 3 K | Reserved | |
| | 0x4002 2000 - 0x4002 23FF | 1 K | Flash interface | Section 3.6 on page 67 |
| | 0x4002 1400 - 0x4002 1FFF | 3 K | Reserved | |
| | 0x4002 1000 - 0x4002 13FF | 1 K | RCC | Section 7.4.14 on page 132 |
| | 0x4002 0800 - 0x4002 0FFF | 2 K | Reserved | |
| | 0x4002 0400 - 0x4002 07FF | 1 K | DMA2 | Section 9.5.7 on page 168 |
| | 0x4002 0000 - 0x4002 03FF | 1 K | DMA1 | |
| | 0x4001 8000 - 0x4001 FFFF | 32 K | Reserved | |
| | 0x4001 4C00 - 0x4001 7FFF | 13 K | Reserved | |
| | 0x4001 4800 - 0x4001 4BFF | 1 K | TIM17 | Section 18.6.17 on page 585 |
| | 0x4001 4400 - 0x4001 47FF | 1 K | TIM16 | |
| | 0x4001 4000 - 0x4001 43FF | 1 K | TIM15 | Section 18.5.18 on page 567 |

**(Figura 11)**

Cerchiata in rosso vediamo la sigla che stavamo cercando, mentre il rettangolo rosso rappresenta l'indirizzo base della periferica RCC. A breve questa informazione ci sarà molto utile.

### 2.4.3 Registri GPIO, Una Breve Panoramica

E' arrivato il momento di riprendere il concetto di registri GPIO, di cui ne abbiamo dato un breve accenno nell'introduzione di questo capitolo. Ogni periferica di I/O sulla board è gestita da un banco di registri. Sulla nostra board abbiamo 6 porte **GPIOx** ($x$= $A,B,C,D,E,F$) a cui è associato un **set di registri a 32-bit**.

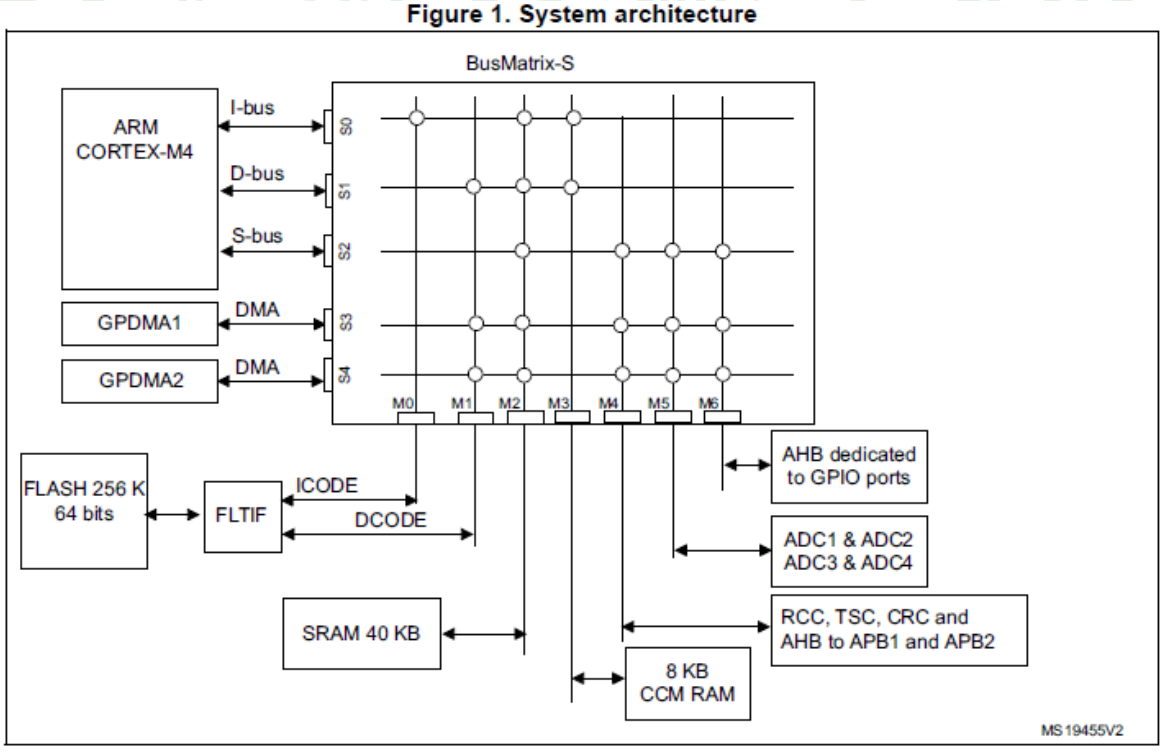
In particolare abbiamo quattro registri di configurazione a 32-bit (**GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR e GPIOx_PUPDR**), due registri dati a 32-bit (**GPIOx_IDR e GPIOx_ODR**), un registro set/reset a 32-bit (**GPIOx_BSRR**), un registro di blocco a 32-bit (**GPIOx_LCKR**) e 2 registri di selezione alternate function (**GPIOx_AFRH e GPIOx_AFRL**).

Incominciamo quindi a capire che la sigla referente al led blu, ovvero PE8, è in realtà riferita al bit 8 del registro GPIOE. Ma proseguiamo con ordine, una volta trovata la periferica di interesse, dobbiamo trovare il **bus** a cui si riferisce, ovvero il bus che controlla le GPIO, questo perché ci sarà d'aiuto nel trovare l'indirizzo base della GPIOE, ma soprattutto perché permetterà di orientarci su quale registro della periferica RCC dobbiamo lavorare per abilitare il clock dei diversi led scelti.

### 2.4.4 Ricerca del Bus di Riferimento

Per trovare il bus di riferimento ci torna utile in questo momento la seguente tabella del Reference Manual:

**Figure 1. System architecture**




**(Figura 12)**

Abbiamo inserito uno schema analogo nel primo capitolo, in particolare nel paragrafo 1.3, dove abbiamo parlato dell'architettura del microcontrollore. La figura che vediamo ora in **Figura 12**, rappresenta una matrice di interconnessioni tra i 5 master a sinistra e i 7 slave in basso, noi siamo interessati però ai bus, in particolare a quello che collega le GPIO al sistema, ci

orientiamo quindi sulla parte dei 7 slaves e troviamo sull'estrema destra la seguente voce "**AHB dedicated to GPIO ports**". Capiamo quindi che il bus **AHB** è il nostro bus di interesse. Ritorniamo alla **Figura 11** che mostra l'elenco di tutti gli indirizzi base referenti a ciascuna periferica; osserviamo quindi che la prima colonna fa riferimento ai BUS, cerchiamo quindi tra le periferiche che presentano il bus **AHB** e tra quelle referenti alla sezione **AHB2** troviamo la GPIOE:

**Table 2. STM32F30xx memory map and peripheral register boundary addresses**

| Bus | Boundary address | Size (bytes) | Peripheral | Peripheral register map |
|---|---|---|---|---|
| AHB3 | 0x5000 0400 - 0x5000 07FF | 1 K | ADC3 - ADC4 | *Section 12.14.4 on page 299* |
| | 0x5000 0000 - 0x5000 03FF | 1 K | ADC1 - ADC2 | |
| | 0x4800 1800 - 0x4FFF FFFF | ~132 M | Reserved | |
| AHB2 | 0x4800 1400 - 0x4800 17FF | 1 K | GPIOF | *Section 8.4.12 on page 149* |
| | 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 | |
| | 0x4002 4400 - 0x47FF FFFF | ~128 M | Reserved | |

**(Figura 13)**

Nel rettangolo di colore rosso troviamo l'indirizzo di cui abbiamo bisogno.

### 2.4.5 Registri RCC

Sempre attraverso il Reference Manual ora ci spotiamo nella sezione "**Reset an clock control (RCC)**", arriviamo quindi al sottopragrafo "**RCC registers**", dato che noi siamo interessati a modificare i registri della periferica RCC per abilitare il clock della porta GPIOE, quindi al sottoparagrafo "**AHB peripheral clock enable register (RCC_AHBENR)**". Questo perché come detto in precedenza siamo interessati a controllare il clock di una periferica presente sul bus AHB.

### 7.4.6 AHB peripheral clock enable register (RCC_AHBENR)

Address offset: 0x14

Reset value: 0x0000 0014

Access: no wait state, word, half-word and byte access

*Note: When the peripheral clock is not active, the peripheral register values may not be readable by software and the returned value is always 0x0.*

| 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 |

**(Figura 14)**

Osserviamo da subito l'**Address Offset**. Questo valore (in questo caso 0x14) deve essere sommato all'indirizzo base trovato precedentemente per quanto riguardo la periferica RCC (ricordiamo quindi l'indirizzo 0x40021000) per puntare correttamente al registro che gestisce l'RCC del bus AHB, precisamente al registro da 32 bit mostrato di seguito nella figura. Osserviamo che alcuni bit di questo registro gestiscono una particolare periferica, ovvero abilitano il clock della stessa; quella sottolineata in tondo rappresenta appunto il bit 21 a cui fa riferimento **IOPEEN** (acronimo di I/O Ports E ENable), quindi portando ad 1 quel bit siamo sicuri di aver abilitato il clock del registro GPIOE; siamo pronti ora a scrivere le prime linee di codice. Avviamo IAR Embedded Workbench for ARM e creiamo un progetto vuoto come visto nel Capitolo 1, scriviamo quindi nel **main.c** le seguenti istruzioni:

main.c *

```
#define RCC_AHBENR      (unsigned int*)      (0x40021014) //INDIRIZZO BASE DEL REGISTRO RCC RIFERITO AL
                                                          //BUS AHB (VALORE 0x40021000 + 0x14

#define IOPE_EN         (unsigned int)       (1<<21)      //SEQUENZA DI 32 BIT IN CUI SONO TUTTI 0 E SOLO
                                                          //IL BIT 21 E' PARI AD 1, SENZA MODIFICARE
                                                            //NESSUN ALTRO BIT DEL REGISTRO DI INTERESSE


void main(void)
{
  unsigned int* puntatore;

  puntatore = RCC_AHBENR;
  *puntatore |= (IOPE_EN);
```

Attraverso i commenti del codice, possiamo renderci conto del significato delle prime due istruzioni. All'interno del main invece abbiamo creato una variabile a puntatore, l'abbiamo eguagliata all'indirizzo che punta al registro RCC e successivamente abbiamo fatto l'**OR** del registro puntato dal puntatore e quella sequenza di bit precedentemente definita con nome di **IOPE_EN**. Questo procedimento verrà ripetuto ogni volta che si vorrà cambiare un bit all'interno di un registro, con l'unica differenza che se si vorrà inserire uno zero nella sequenza allora si farà la **AND** con una sequenza di bit posti tutti ad 1 tranne il bit di interesse a 0.

Abilitato il clock è necessario ora caratterizzare la porta GPIOE.

### 2.4.6 Registri GPIO, Configurazione

Andiamo al capitolo del Reference Manual intitolato "**General Purpose I/Os"**, quindi al sottoparagrafo referente ai registri delle GPIO, ovvero il sottopragrafo intitolato "**GPIO registers".** Qui troviamo tutti i registri al corretto offset dall'indirizzo di base che vanno a caratterizzare la porta, portiamoci quindi alla voce "**GPIO port mode register (GPIOx_MODER) (x = A..F)"**. Quest'ultimo è un registro a 32-bit che serve per settare la modalità di ciascun pin della porta "x" (x = A..F) che in generale si divide in due gruppi **GPIO** ed **EXTI**. Alla categoria GPIO appartengono modalità come **Input, Output, Alternate Function e Analog Mode**, mentre ad EXTI si riferiscono tutte le modalità che vedono GPIO come "elemento" interrompente (External Interrupts).

**8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..F)**

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

| 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 |

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

**(Figura 15)**

Abbiamo capito in precedenza a cosa ci servono dati come l'address offset. Ora nello specifico dobbiamo selezionare i bit del registro che fanno riferimento ai led che vogliamo accendere e settarli con i giusti valori. Noi vogliamo porre i bit relativi ai pin dal 15 al 9 (i pin realitvi ai led della board) in modalità di uscita ossia "**General purpose output mode**".

Non ci basterà altro che inserire le seguenti righe di codice:

main.c *

```
#define RCC_AHBENR      (unsigned int*)         (0x40021014) //INDIRIZZO BASE DEL REGISTRO RCC RIFERITO AL
                                                              //BUS AHB (VALORE 0x40021000 + 0x14

#define IOPE_EN         (unsigned int)          (1<<21)      //SEQUENZA DI 32 BIT IN CUI SONO TUTTI 0 E SOLO
                                                              //IL BIT 21 E' PARI AD 1, SENZA MODIFICARE
                                                                //NESSUN ALTRO BIT DEL REGISTRO DI INTERESSE

#define GPIOE_MODER     (unsigned int*)         (0x48001000) //INDIRIZZO DI BASE DELLA NOSTRA PORTA GPIOE
                                                              //RIFERITO AL REGISTRO MODER


void main(void)
{
  unsigned int* puntatore;

  puntatore = RCC_AHBENR;
  *puntatore |= (IOPE_EN);

  puntatore = GPIOE_MODER;
  *puntatore |= 0x55550000;            //General Purpose Output Mode pins 15-9
```

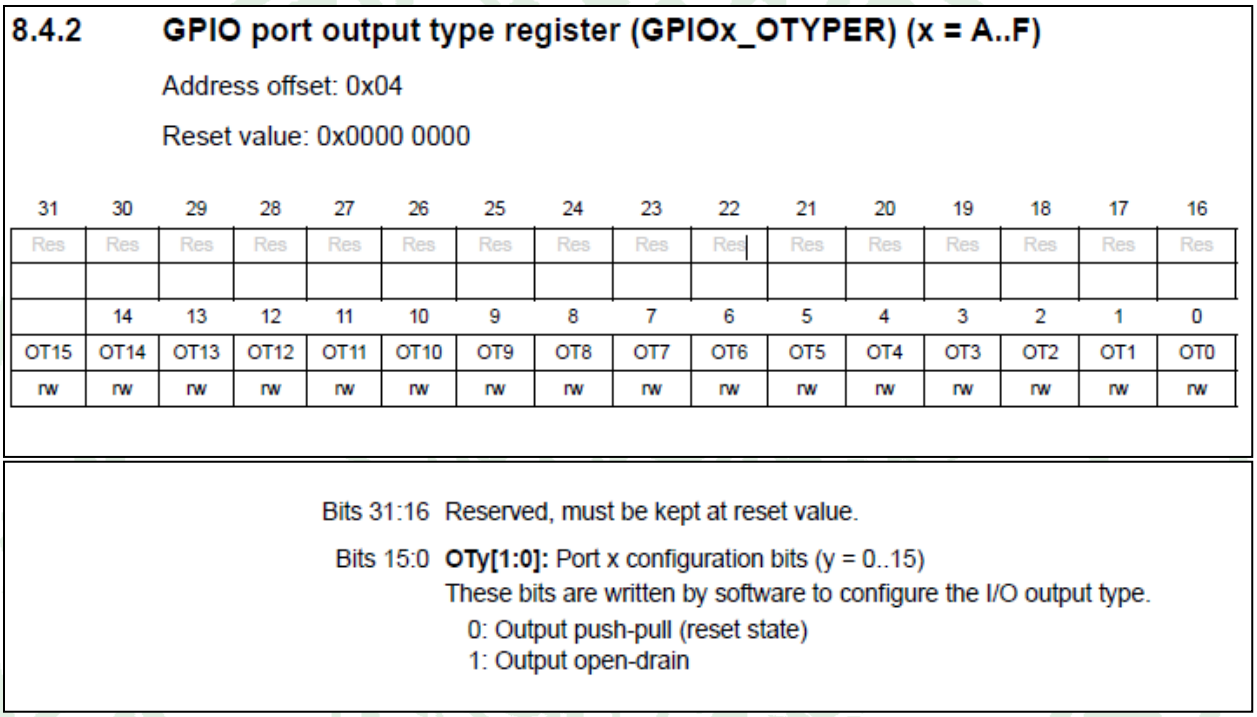
Così facendo abbiamo nuovamente creato una variabile puntatore che fa riferimento all'indirizzo di base della nostra porta GPIOE, in particolare al registro moder che aveva offset uguale a 0x00, quindi uguale all'indirizzo stesso della porta GPIOE. Successivamente abbiamo fatto la OR del registro puntato dal puntatore e la sequenza di bit 0000 0000 0101 0101 0101 0101 0000 0000 0000 0000 che ci permetterà di configurare i nostri pin di interesse in modalità General Purpose Output.

Il procedimento ora risulta chiaro per tutti i setting successivi, ovvero definisco un indirizzo che punta al registro di interesse per il setting della GPIOE e successivamente mi definisco le sequenze che interessano al setting dei pin di interesse, quindi andrò a fare la OR se voglio inserire 1, altrimenti la AND col negato se voglio inserire 0.

Ora passiamo alla configurazione dell'**Output Type Register** (**GPIOx_OTYPER**) **(x = A..F)** ed impostiamo l'Output Type, ovvero dobbiamo specificare se l'output deve essere collegato al **Vdd** tramite un MOS o ad una tensione negativa **Vs** sempre tramite un MOS.

Nel primo caso diremo che che l'output type è messo in **push-pull**, mentre nel secondo caso in **open drain**.

Scegliamo la prima opzione quindi:



**8.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..F)**

Address offset: 0x04

Reset value: 0x0000 0000

| 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 |
| | | | | | | | | | | | | | | | |
| | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OT15 | OT14 | OT13 | OT12 | OT11 | OT10 | OT9 | OT8 | OT7 | OT6 | OT5 | OT4 | OT3 | OT2 | OT1 | OT0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output type.
0: Output push-pull (reset state)
1: Output open-drain

**(Figura 16)**

Quindi nel registro all'offset 0x04 dall'indirizzo base di quello della GPIOE settiamo i bit da 0 a 15 al valore 0.

Proseguiamo con l'**Output Speed Register (GPIOx_OSPEEDR) (x = A..F)** e con il **Pull-up/Pull-down register (GPIOx_PUPDR) (x = A..F)** che vengono utilizzati rispettivamente per settare la velocità del clock per questa periferica e per far si che l'uscita passi attraverso un MOS di pull-up o pull-down. Riguardo il primo settaggio non siamo interessati ad una velocità di clock in particolare, porremo quindi i bit nella configurazione "**Low Speed**", per il secondo settaggio invece non saremo interessati ne a farlo passare per un MOS di pull-up ne per uno di pull-down, quindi sceglieremo la configurazione "**No pull-up No pull-down**", vediamo come:

### 8.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A..F)

Address offset: 0x08

Reset value:

- 0x6400 0000 for port A
- 0x0000 00C0 for port B
- 0x0000 0000 for other ports

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OSPEEDR15[1:0] | | OSPEEDR14[1:0] | | OSPEEDR13[1:0] | | OSPEEDR12[1:0] | | OSPEEDR11[1:0] | | OSPEEDR10[1:0] | | OSPEEDR9[1:0] | | OSPEEDR8[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 |
| OSPEEDR7[1:0] | | OSPEEDR6[1:0] | | OSPEEDR5[1:0] | | OSPEEDR4[1:0] | | OSPEEDR3[1:0] | | OSPEEDR2[1:0] | | OSPEEDR1[1:0] | | OSPEEDR0[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 2y+1:2y **OSPEEDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output speed.
x0: 2 MHz Low speed
01: 10 MHz Medium speed
11: 50 MHz High speed
*Note: Refer to the device datasheet for the frequency specifications and the power supply and load conditions for each speed.*

**(Figura 17)**

### 8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..F)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0C00 0000 for other ports

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PUPDR15[1:0] | | PUPDR14[1:0] | | PUPDR13[1:0] | | PUPDR12[1:0] | | PUPDR11[1:0] | | PUPDR10[1:0] | | PUPDR9[1:0] | | PUPDR8[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 |
| PUPDR7[1:0] | | PUPDR6[1:0] | | PUPDR5[1:0] | | PUPDR4[1:0] | | PUPDR3[1:0] | | PUPDR2[1:0] | | PUPDR1[1:0] | | PUPDR0[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

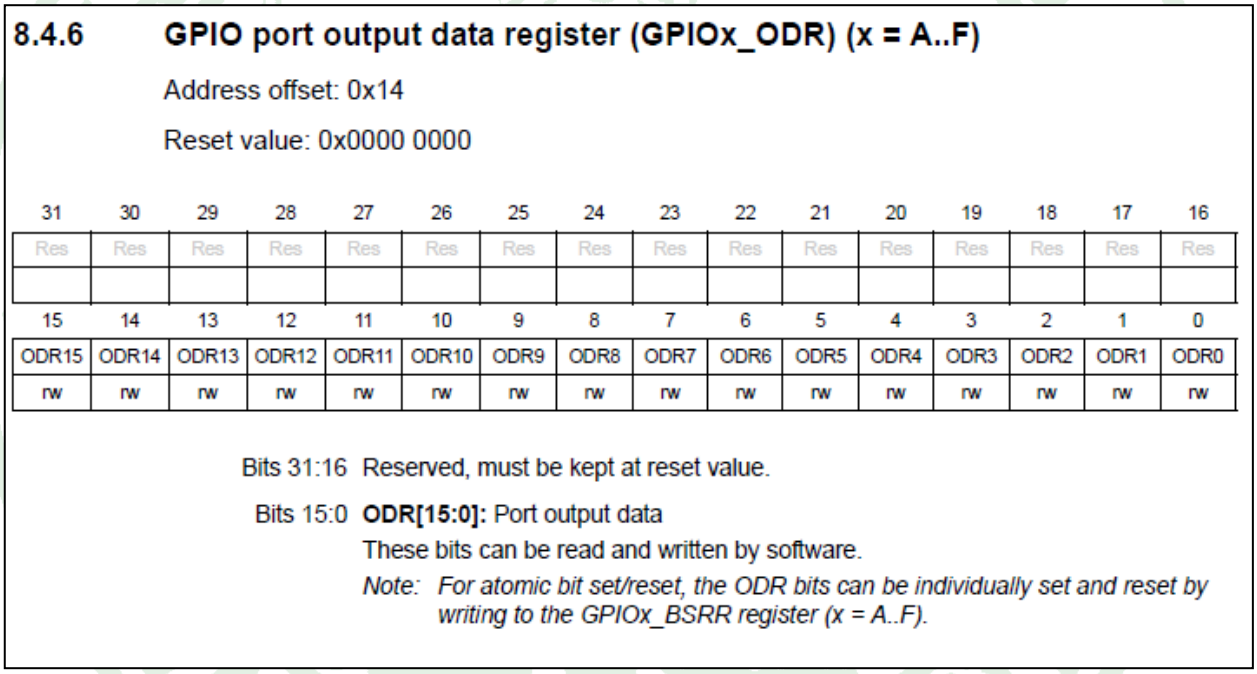
Bits 2y+1:2y **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

**(Figura 18)**

Con questo passaggio si conclude il settaggio dei pin PE15-9, non resta che abilitarli, ovvero inviare un bit 1 in modo che questi si accendano; si ribadisce che tutti i passaggi precedenti non servivano ad altro che per **specificare** i settaggi della periferica. Quest'ultimo passaggio prevede invece direttamente l'**abilitazione** della periferica, in questo caso l'accensione dei led.

Il registro di nostro interesse in questo momento è **l'Output Data Register (GPIOx_ODR) (x= A..F)**. Risulta chiaro, capito il meccanismo che per abilitare i led dobbiamo portare ad 1 i bit da 0 a 15, così come mostrato in precedenza, c'è però da specificare un percorso alternativo.

Prima di mostrarlo occorre spiegare che cosa vuol dire "rw" al di sotto di ciascuna casella.



**8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..F)**

Address offset: 0x14

Reset value: 0x0000 0000

| 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 |

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODR[15:0]:** Port output data
These bits can be read and written by software.
*Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..F).*

**(Figura 17)**

"rw" sta per **read/write** ovvero specifica che per questo bit posso sia leggere il valore al proprio interno sia scriverci sopra, ci saranno però casi in cui venga indicato solo "w" ovvero solo write. Quelli saranno bit in cui è possibile solo scrivere un dato, ma non leggerlo. Possiamo immaginare nel primo caso una configurazione **latch**, mentre nel secondo una configurazione che prevede solo il cambio di valore all'interno del bistabile, ma che non ne tenga "**memoria**" del valore cambiato.

Fatta questa premessa vediamo ora il **Bit Set/Reset (GPIOx_BSRR) (x= A..F)**.

### 8.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..F)

Address offset: 0x18

Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| w | w | w | w | 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 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
*Note: If both BSx and BRx are set, BSx has priority.*

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

**(Figura 18)**

Questo registro rispetto al precedente prevede solo di scrivere un dato, non di leggerlo; però permette in un solo registro di settare e resettare il dato da trasferire alla periferica. Infatti i primi 16 bit **settano** uno dei 16 bit interessati della GPIO scelta, mentre gli altri 16 lo **resettano**. I due metodi sono del tutto equivalenti per poter abilitare i nostri led, nel codice che seguirà useremo però il registro **GPIOE_ODR**.

## 2.4.7 Codice Finale

Il codice finale del nostro progetto, viste tutte le considerazioni fatte fino ad ora, sarà il seguente :

main.c

```
#define RCC_AHBENR      (unsigned int*)         (0x40021014) //INDIRIZZO BASE DEL REGISTRO RCC RIFERITO AL
                                                              //BUS AHB (VALORE 0x40021000 + address offset)
                                                                //address offset 0x14

#define IOPE_EN         (unsigned int)          (1<<21)      //SEQUENZA DI 32 BIT IN CUI SONO TUTTI 0 E SOLO
                                                              //IL BIT 21 E' PARI AD 1, SENZA MODIFICARE
                                                                //NESSUN ALTRO BIT DEL REGISTRO DI INTERESSE

#define GPIOE_MODER     (unsigned int*)         (0x48001000) //INDIRIZZO DI BASE DELLA NOSTRA PORTA GPIOE
                                                              //RIFERITO AL REGISTRO MODER
                                                                //address offset 0x00

#define GPIOE_OTYPER    (unsigned int*)         (0x48001004) //INDIRIZZO DI BASE DELLA NOSTRA PORTA GPIOE
                                                              //RIFERITO AL REGISTRO OTYPER
                                                                //address offset 0x04

#define GPIOE_OSPEEDR   (unsigned int*)         (0x48001008) //INDIRIZZO DI BASE DELLA NOSTRA PORTA GPIOE
                                                              //RIFERITO AL REGISTRO OSPEEDR
                                                                //address offset 0x08

#define GPIOE_ODR       (unsigned int*)         (0x48001014) //INDIRIZZO DI BASE DELLA NOSTRA PORTA GPIOE
                                                              //RIFERITO AL REGISTRO ODR
                                                                //address offset 0x14

#define GPIOE_PUPDR     (unsigned int*)         (0x4800100C) //INDIRIZZO DI BASE DELLA NOSTRA PORTA GPIOE
                                                              //RIFERITO AL REGISTRO PUPDR
                                                                //address offset 0x0C

void main(void)
{
  unsigned int* puntatore;

  puntatore = RCC_AHBENR;
  *puntatore |= (IOPE_EN);

  puntatore = GPIOE_MODER;
  *puntatore |= 0x55550000;             //General Purpose Output Mode pins 15-9

  puntatore = GPIOE_OTYPER;
  *puntatore |= 0;                      //Push-Pull Output Type

  puntatore = GPIOE_OSPEEDR;
  *puntatore |= 0;                      //Low Speed

  puntatore = GPIOE_PUPDR;              //No Pullup/No Pulldown
  *puntatore |= 0;

  puntatore = GPIOE_ODR;

 //VARIABILI
 int count = 1<<8;
 int i;
 int tempo = 720000;

 while (1)
 {
     //ACCENSIONE DI TUTTI I LED
     while (count <= (15<<8 | 15<<12)) //cicla finchè non restano tutti i led accesi
     {
       *puntatore = count;
       count += (1<<8); //somma in esadecimale => accendo anche il led successivo

       for (i=0; i<tempo; i++) { //lettura/scrittura avvengono ad ogni colpo di clock AHB, cioè ogni 1/72 us = 0,014 us = 14 ns
       }
     }
   //SPEGNIMENTO DI TUTTI I LED
     while (count >= 0)
     {
       *puntatore = count;
        count -= (1<<8);

       for (i=0; i<tempo; i++){
       }
     }
 }

}  // end main
```

Abbiamo inserito un ciclo while per permettere l'accensione di tutti i led.

Il ciclo for all'interno del ciclo while invece ci serve per far si che sia lettura che scrittura avvengano ad ogni colpo di clock AHB.

Una variante di questo esempio potrebbe essere quella con l'utilizzo dei timer invece di usare una forzatura come quella che può essere il ciclo for per garantirci un certo tempo d'attesa.

Nel successivo capitolo quindi affronteremo l'argomento dei timer, in parciolar modo quello dei timer general purpose.

# Capitolo 3: Timer

I **General Purpose Timers** sono una tipologia di Timer Hardware presenti sulle board STM32. Rispetto ai Basic Timer, introducono ulteriori funzionalità grazie all'hardware aggiuntivo. Possono essere utilizzati per una varietà di scopi, dal semplice conteggio al calcolo della **PWM** (Pulse Width Modulation) di un segnale in input oppure nella generazione di diverse forme d'onda. I General Purpose Timers sono realizzati in hardware completamente indipendente dal processore e il loro utilizzo non comporta alcun **overhead** (richiesta di risorse in eccesso rispetto a quelle strettamente necessarie).

## 3.1 Caratteristiche Principali

Le principali caratteristiche dei General-Purpose Timers comprendono:

- up, down, up/down auto-reload counter
- 4 canali indipendenti per:
    - Input capture
    - Output compare
    - PWM generation (Edge- and Center-aligned modes)
    - One-pulse mode output
- la possibilità di interconnettere più timer
- supporto incremental (quadrature) encoder e hall-sensor interface per il posizionamento.
- può genereare un interruzione DMA

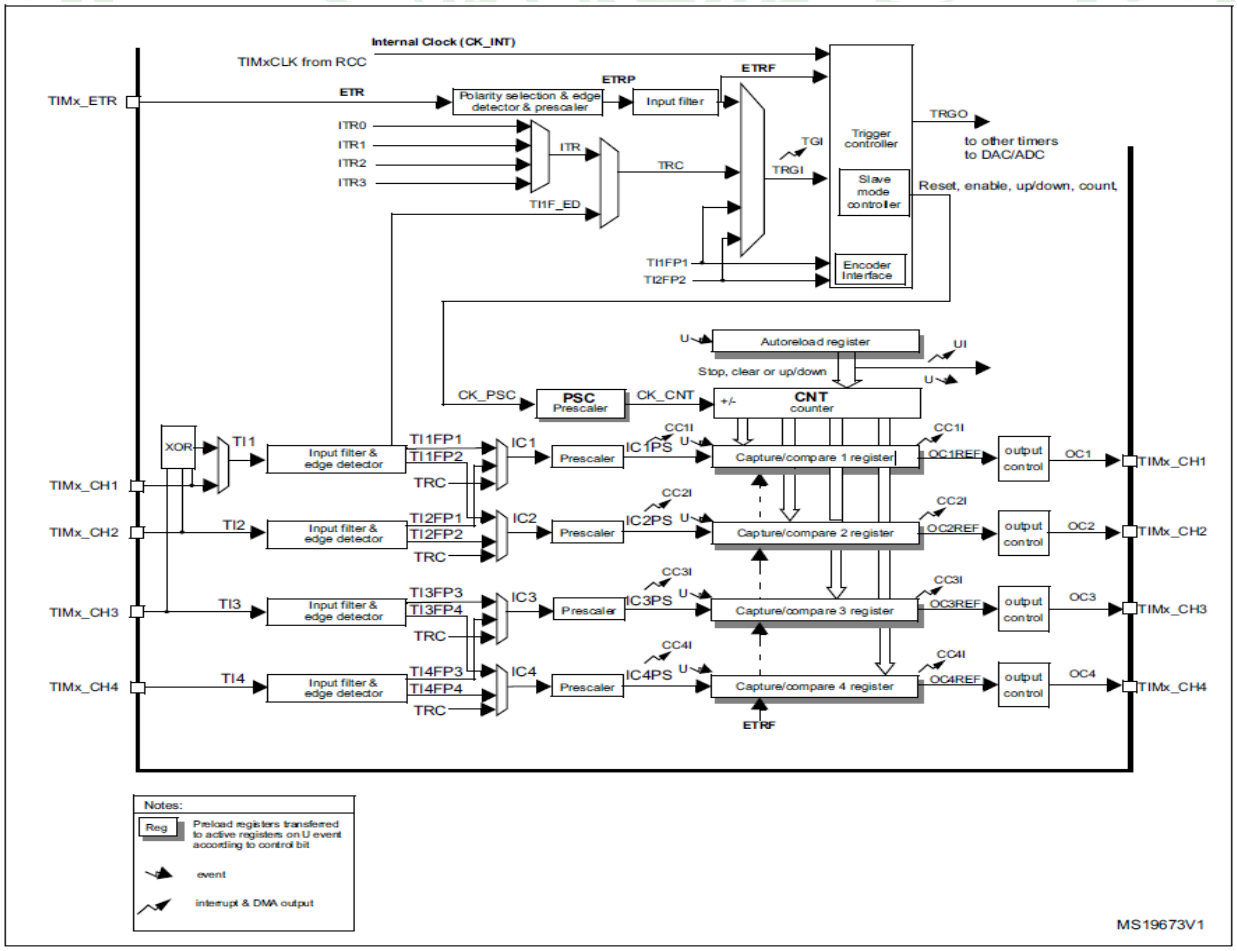
## 3.2 Struttura

Come per i Basic Timer la time base unit comprende:

- un contatore a 16/32 bit
- un prescaler a 16 bit
- Circuito di sincronizzazione con i DAC
- un registro di auto-reload
- un registro di conteggio corrente
- un registro per il prescaler

in aggiunta troviamo:

- un unità di cattura/comparazione
- un encoder interface
- circuito di sincronizzazione Timers
- hall sensor interface




**(Figura 19)**

## 3.3 Funzionamento di Base

Il contatore può essere configurato per un **conteggio** ad **incremento**, **decremento** o **centrale** (incrementa e poi decrementa). Quando il contatore è a incremento genera un segnale di overflow nel momento in cui raggiunge il valore contenuto nell'auto-reload register al quale ne consegue un UEV. Viceversa se il contatore è a decremento genera un UEV al raggiungimento dell'underflow. Per il conteggio centrale, invece, viene generato un segnale di overflow al raggiungimento della soglia e uno di underflow al raggiungimento del valore 0.

Il registro di autoreload non subisce modifiche ma bensì le operazioni di lettura/scrittura dal registro avvengono attraverso un preload register.

Il prescaler a 16-bit può dividere la frequenza di clock di un fattore che va da 1 a 65536. Può essere modificato al volo durante il funzionamento dove il nuovo valore sarà valido al successivo UEV.

Quando si genera un evento di update tutto i registri vengono aggiornati e il flag UIF viene settato alto:

- Il buffer del prescaler è ricaricato al valore contenuto nel registro del prescaler
- Il registro di auto-reload viene ricaricato con il valore contenuto nel preload register. Nota: se il valore del auto-reload register è stato modificato verrà caricato il nuovo valore.

Ogni timer inoltre supporta sia le interrupt che il DMA.

# Conclusioni

In conclusione abbiamo visto quanto sia vasto il mondo dei microcontrollori, e quanto soprattutto sia in continua evoluzione. Il loro grande successo mette costantemente in competizione le più grandi aziende produttrici per offrire al pubblico prodotti sempre più performanti ed a basso costo. Successo che è stato dato come abbiamo potuto constatare dalla facilità di utilizzo, di programmabilità e soprattutto dalla convenienza economica rispetto a sistemi meno compatti e ad uso più generico, di sicuro molto più costosi. I settori di utilizzo svariati permetteno di affacciarsi in una vasta gamma di applicazioni. Certo il loro campo di utilizzo, come abbiamo visto, è molto specifico e per questo risulta essere limitato per quanto riguarda invece un uso generico, ma resta pur vero che il compito per cui viene progettato, viene svolto nel modo più efficiente possibile.

Insomma un circuito integrato universale, economico, performante, che può essere programmato ed usato in svariati campi dell'elettronica, per ogni necessità e bisogno, che in modo invisible ci accompagna nella vita quotidiana, dentro e fuori le nostre case, senza rendercene conto.

Questo è un microcontrollore.

# Bibliografia

[1] STMicroelectronics , RM0316 Reference Manual

[2] STMicroelectonics, UM1570 User Manual

[3] Hitex Development Tools, The Insider's Guide To The STM32 ARM® Based Microcontroller

[4] La Storia del Microcontrollore, http://it.wingwit.com/hardware/computer-drives-storage/47399.html, 25/07/2015

[5] Watchdog, http://www.webopedia.com/TERM/W/watchdog.html, 02/09/2015