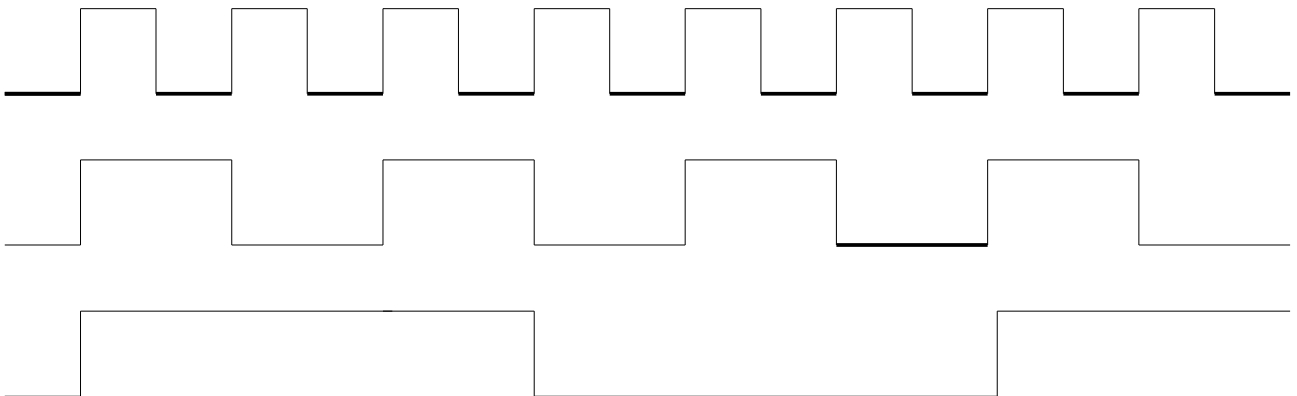


# ***C18 Step by Step***

*Imparare a programmare i PIC 18*



# Informativa

Come prescritto dall'art. 1, comma 1, della legge 21 maggio 2004 n.128, l'autore avvisa di aver assolto, per la seguente opera dell'ingegno, a tutti gli obblighi della legge 22 Aprile del 1941 n. 633, sulla tutela del diritto d'autore.

Tutti i diritti di questa opera sono riservati. Ogni riproduzione ed ogni altra forma di diffusione al pubblico dell'opera, o parte di essa, senza un'autorizzazione scritta dell'autore, rappresenta una violazione della legge che tutela il diritto d'autore, in particolare non ne è consentito un utilizzo per trarne profitto.

La mancata osservanza della legge 22 Aprile del 1941 n. 633 è perseguibile con la reclusione o sanzione pecuniaria, come descritto al Titolo III, Capo III, Sezione II.

A norma dell'art. 70 è comunque consentito, per scopi di critica o discussione, il riassunto e la citazione, accompagnati dalla menzione del titolo dell'opera e dal nome dell'autore.

# Avvertenze

Chiunque decida di far uso delle nozioni riportate nella seguente opera o decida di realizzare i circuiti proposti, è tenuto a prestare la massima attenzione in osservanza alle normative in vigore sulla sicurezza.

L'autore declina ogni responsabilità per eventuali danni causati a persone, animali o cose derivante dall'utilizzo diretto o indiretto del materiale, dei dispositivi o del software presentati nella seguente opera.

Si fa inoltre presente che quanto riportato viene fornito così com'è, a solo scopo didattico e formativo, senza garanzia alcuna della sua correttezza.

Tutti i marchi citati in quest'opera sono dei rispettivi proprietari.

# Licenza Software

---

Tutto il Software presentato in questo testo è rilasciato sotto licenza GNU, di seguito riportata. Le librerie utilizzate potrebbero essere soggette a differente licenza dunque per un utilizzo corretto del software e sua redistribuzione far riferimento alle particolari librerie incluse.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

#### **Additional Terms**

As stated in section 7 of the license, these additional terms are included:

- 1) The copyright statement must be preserved.
- 2) Any Warranty Disclaimers must be preserved.
- 3) The name of the Authors and Contributors can not be used for publicity purposes.

---

# Presentazione

Dopo tre anni dalla pubblicazione della prima edizione “C18 Step by Step” mi sono convinto di aggiornare il documento. Il successo mostrato dai numerosi download giornalieri e dai feedback dei lettori, mi hanno portato a rivedere la prima versione, nata con l'intento di riassumere l'utilizzo del compilatore C18 e l'ambiente di sviluppo MPLAB. Nonostante il successo della prima edizione, ero consapevole di suoi molti limiti educativi legati al fatto che la prima edizione aveva quasi le vesti di un manuale degli appunti. Il lettore meno esperto, anche se motivato, incontrava spesso delle difficoltà associate effettivamente alla mancanza di spiegazioni. In questa seconda edizione, forte dei feedback dei lettori, nonché consapevole delle difficoltà incontrate dagli stessi, il testo è stato interamente riveduto e aggiornato. Nuovi Capitoli introduttivi sono stati scritti per colmare le problematiche che i lettori meno esperti hanno incontrato, rendendo questa seconda edizione un testo ideale per tutti i lettori che si avvicinano anche per la prima volta al mondo dei microcontrollori.

Nonostante i Capitoli introduttivi, una certa familiarità da parte del lettore con l'utilizzo del PC e conoscenze base dell'elettronica, è richiesta. Il linguaggio C viene introdotto in maniera tale che anche coloro che non hanno familiarità con lo stesso, possano facilmente apprenderlo. Se il lettore ha conoscenze di altri linguaggi sarà sicuramente agevolato.

Il testo è scritto con particolare cura e attenzione per applicazioni didattiche. Molti professori hanno infatti utilizzato la prima versione come guida ai propri corsi interni sui microcontrollori e programmazione di sistemi embedded, quali per esempio corsi di robotica. Grazie ai nuovi Capitoli introduttivi i docenti, nonché gli studenti, siano essi di scuole superiori od universitari, avranno la possibilità di effettuare lo studio in maniera più indipendente e completa. A supporto del testo e a scopi didattici è stata anche progettata la nuova scheda di sviluppo Freedom II, per mezzo della quale è possibile fare tutte le esperienze descritte nel testo, senza avere l'esigenza di aggiungere hardware esterno.

Il materiale per agevolare la realizzazione della scheda di sviluppo può essere inoltre richiesto, previa donazione di supporto, alla sezione Servizi del sito [www.LaurTec.it](http://www.LaurTec.it). Dallo stesso sito è possibile scaricare tutta la documentazione tecnica associata alla scheda Freedom II.

I ricavati delle donazioni sono utilizzati per lo sviluppo e la realizzazione di nuovi progetti e testi scaricabili gratuitamente e fruibili dalla comunità. Naturalmente ogni altra scheda di sviluppo può essere utilizzata per scopi didattici, vista la generalità delle applicazioni.

Si fa presente che la descrizione del modulo USB è volutamente non trattato in questa edizione del testo vista la maggiore esperienza richiesta per l'utilizzo della porta USB. Ciononostante dal sito [www.LaurTec.it](http://www.LaurTec.it) è possibile scaricare altra documentazione specifica per l'utilizzo della porta USB. In particolare il progetto EasyUSB (PJ7008) permette lo sviluppo di applicazioni ed interfacce facilitando l'apprendimento della porta USB.

Vorrei cogliere l'occasione di ringraziare i numerosi lettori che con il loro feedback ed entusiasmo mi hanno spinto a scrivere questa seconda edizione del testo. Vorrei inoltre ringraziare la comunità di “robottari” italiani che è possibile trovare ogni giorno al sito [www.roboitalia.com](http://www.roboitalia.com). Incessabili creatori di macchine futuristiche...e sempre pronti a ritrovarsi in ristoranti...

Con la speranza di aver fatto un buon lavoro, vi auguro una buona lettura.

...come sempre sono in attesa dei vostri commenti.

...per stimolare una terza edizione.

---

## Com'è organizzato il testo

La seconda edizione “C18 Step by Step” è organizzata in modo da guidare passo passo il lettore, nello sviluppo dei propri sistemi embedded. Il testo assume ad ogni passo che lettore sappia quanto spiegato precedentemente, dunque è consigliabile che la lettura avvenga seguendo l'ordine dei Capitoli proposto. Tutto il software è scritto in maniera tale che possa essere facilmente copiato ed incollato nel proprio progetto. Tutti i progetti sono inoltre raccolti e scaricabili direttamente dal sito [www.LaurTec.it](http://www.LaurTec.it).

I vari Capitoli possono essere raggruppati nel seguente modo:

### **Capitolo I, II, III,**

In questi Capitoli vengono introdotti i concetti base dei PIC18 e gli strumenti necessari per lo sviluppo di applicazioni embedded. In particolare viene introdotta l'architettura dei PIC18 in maniera da poter affrontare le applicazioni descritte nei Capitoli successivi.

### **Capitolo IV, V,**

Il Capitolo IV mostra un primo esempio di programmazione senza entrare nel vivo delle potenzialità dei PIC18, mostrando piuttosto la semplicità d'uso. Il Capitolo V mostra come simulare i programmi senza necessariamente necessitare di una scheda di sviluppo per il test. In questo Capitolo sono introdotti anche alcuni concetti per il Debug.

### **Capitolo VI,**

In questo Capitolo sono introdotti tutti gli aspetti fondamentali della programmazione in C focalizzando il suo utilizzo alla programmazione dei PIC18. Numerosi esempi sono utilizzati per ogni istruzione. La spiegazione descrive anche le possibilità di ottimizzazione del codice, mostrando in particolare l'utilità di scrivere un codice chiaro piuttosto che ottimizzato e difficile da leggere. In particolare viene spesso messo in evidenza che un codice apparentemente “ottimizzato” non necessariamente è un codice più veloce!

### **Capitolo VII, VIII, IX, X, XI, XII, XIII**

Ogni Capitolo affronta una particolare periferica interna ed esterna al PIC, il cui utilizzo permette di realizzare applicazioni complesse e professionali. Ogni Periferica trattata è accompagnata dalla spiegazione dell'hardware e sue impostazioni, vengono inoltre introdotte le librerie Microchip e quelle LaurTec appositamente scritte per controllare l'Hardware presente sulla scheda Freedom II.

---

*E' il tempo di pensare a chi dedicare le pagine che seguiranno queste parole. Credo che non bisogna forse pensare troppo che se questo libro è nato è legato al fatto che io stesso sia nato e cresciuto con i giusti stimoli ed interessi, dunque non posso non dedicare questo libro ai miei familiari, papà, mamma e i miei tre fratelli e singola sorellina. Vorrei fare anche un passo in alto e ricordare mio nonno Adolfo che tanto mi ha stimato per i traguardi raggiunti ma che sfortunatamente non ha avuto modo di vedere il tempo della mia laurea e quello che sarebbe seguito.*

*Non vorrei dimenticare gli altri miei nonni i cui ricordi sono però offuscati dalla giovane età in cui li ho conosciuti e visti per l'ultima volta...ma ero il loro Bobo.*

*"Last but not least" un pensiero al mio amore che mi sopporta da un po' di tempo...ma la vita è ancora lunga...!*

*Mauro Laurenti*

---

# Indice

<b>Presentazione.....</b>	<b>4</b>
<b>Com'è organizzato il testo.....</b>	<b>5</b>
<b>Capitolo I.....</b>	<b>9</b>
<b>Un mondo programmabile.....</b>	<b>9</b>
Cos'è un microcontrollore .....	9
I sistemi embedded .....	10
I microcontrollori della Microchip.....	11
Cosa c'è oltre al mondo Microchip.....	12
<b>Capitolo II.....</b>	<b>15</b>
<b>Architettura ed Hardware dei PIC18 .....</b>	<b>15</b>
Architettura dei PIC18 .....	15
Organizzazione della Memoria.....	20
L'oscillatore .....	30
Power Managment .....	35
Circuiteria di Reset.....	38
Le porte d'ingresso uscita.....	42
<b>Capitolo III.....</b>	<b>47</b>
<b>Strumenti per iniziare.....</b>	<b>47</b>
Perché si è scelto Microchip.....	47
L'ambiente di sviluppo MPLAB.....	47
Il compilatore C.....	48
Utilizzare il C o l'Assembler?.....	51
Il programmatore.....	52
Scheda di sviluppo e test .....	53
<b>Capitolo IV.....</b>	<b>55</b>
<b>Salutiamo il mondo per iniziare .....</b>	<b>55</b>
Il nostro primo progetto.....	55
Scrivere e Compilare un progetto .....	63
Programmare il microcontrollore.....	66
Capire come salutare il mondo.....	69
<b>Capitolo V.....</b>	<b>75</b>
<b>Simuliamo i nostri programmi.....</b>	<b>75</b>
Perché e cosa simulare .....	75
Avvio di una sessione di simulazione.....	76
Avvio di una sessione di Debug.....	78
Utilizzo dei Breakpoint.....	80
Controllo delle variabili e registri.....	81
Analisi temporale.....	84
<b>Capitolo VI.....</b>	<b>87</b>
<b>Impariamo a programmare in C.....</b>	<b>87</b>
La sintassi in breve.....	87
Tipi di variabili.....	90
Operatori matematici.....	100
Casting.....	106
Operatori logici e bitwise .....	108
Il ciclo for ( ).....	112
Istruzione condizionale if ( ).....	117

Istruzione condizionale switch ( ).....	124
Istruzione condizionale while ( ).....	129
Le funzioni.....	134
Visibilità delle variabili.....	140
Le interruzioni.....	144
<b>Capitolo VII.....</b>	<b>154</b>
<b>Scriviamo una libreria personale.....</b>	<b>154</b>
Perché scrivere una libreria.....	154
Le direttive.....	156
Esempio di creazione di una Libreria.....	158
Utilizziamo la nostra Libreria.....	163
<b>Capitolo VIII.....</b>	<b>166</b>
<b>Utilizziamo i Timer interni al PIC.....</b>	<b>166</b>
Descrizione dell'hardware e sue applicazioni.....	166
I registri interni per il controllo del Timer0.....	169
Esempi di utilizzo del Timer0.....	170
Impostare il tempo del Timer.....	177
<b>Capitolo IX.....</b>	<b>180</b>
<b>Utilizziamo il modulo PWM interno al PIC.....</b>	<b>180</b>
Descrizione dell'hardware e sue applicazioni.....	180
I registri interni per il controllo del PWM.....	184
Esempio di utilizzo del modulo PWM.....	186
<b>Capitolo X.....</b>	<b>190</b>
<b>Utilizziamo un display alfanumerico LCD.....</b>	<b>190</b>
Descrizione dell'hardware e sue applicazioni.....	190
Utilizzo della libreria LCD.....	193
Leggiamo finalmente Hello World.....	201
<b>Capitolo XI.....</b>	<b>208</b>
<b>Utilizziamo il modulo ADC interno al PIC.....</b>	<b>208</b>
Descrizione dell'hardware e sue applicazioni.....	208
I registri interni per il controllo dell'ADC.....	212
Lettura di una tensione.....	224
Lettura della temperatura.....	227
Lettura dell'intensità luminosa.....	231
<b>Capitolo XII.....</b>	<b>233</b>
<b>Utilizziamo l'EUSART interna al PIC.....</b>	<b>233</b>
Descrizione dell'hardware e sue applicazioni.....	233
Impostare l'EUSART per un corretto utilizzo.....	235
Il codice ASCII.....	242
Esempio di utilizzo dell'EUSART in polling.....	243
Esempio di utilizzo dell'EUSART con interruzione.....	246
<b>Capitolo XIII.....</b>	<b>250</b>
<b>Utilizziamo il modulo I2C interno al PIC.....</b>	<b>250</b>
Descrizione dell'hardware e sue applicazioni.....	250
Impostare il modulo I2C per un corretto utilizzo.....	253
Esempio di lettura di una memoria EEPROM I2C.....	255
Esempio di utilizzo di un Clock Calendar I2C.....	260
<b>Bibliografia.....</b>	<b>265</b>
Internet Link.....	265
Libri.....	265



# Capitolo I

## I

## Un mondo programmabile

Oggigiorno si sente parlare di lavatrici intelligenti, di frigoriferi che si riempiono da soli e giocattoli che parlano. Tutto questo è stato in parte reso possibile grazie ai bassi costi delle logiche programmabili tra queste vi sono i microcontrollori. In questo Capitolo dopo un'introduzione sui microcontrollori e in particolare quelli della Microchip, verranno discusse anche le altre alternative, le quali però non saranno ulteriormente trattate nel seguito del testo.

### Cos'è un microcontrollore

Un microcontrollore o microcalcolatore (MCU), è un circuito integrato ovvero un semiconduttore. Al suo interno sono presenti da poche migliaia a centinaia di migliaia di componenti elettronici elementari; generalmente le tecnologie attuali utilizzano transistor CMOS. Fin qui nulla di nuovo se confrontati con gli altri circuiti integrati, ed in particolare con i microprocessori. In realtà ci sono molte differenze che li rendono appetibili rispetto ai microprocessori. Se consideriamo i microprocessori attualmente sul mercato, per esempio quelli della Intel o AMD, ben sappiamo che hanno potenze di calcolo enormi, non sappiamo quanto, ma sappiamo che possiamo farci molto! In realtà solo con il microprocessore non si può fare nulla! Per poter utilizzare il microprocessore è necessario circondarlo di molto altro Hardware, quale per esempio, la RAM, Hard Disk e di tutte le altre periferiche necessarie (scheda video, Porta Seriale, Porta Ethernet, ecc...). Inoltre il microprocessore con tutto l'armamentario attorno richiede notevole potenza, non a caso sono presenti ventole di raffreddamento!

Bene, al microcontrollore tutto questo non serve, RAM e ROM sono al suo interno, come anche la maggior parte delle periferiche di base o specifiche per determinate applicazioni<sup>1</sup>. In ultimo, ma non meno importante, il microcontrollore assorbe così poco, che alcune applicazioni, tipo orologi digitali, possono essere alimentati per mezzo di un limone usato come batteria. Se il microcontrollore ha tutti questi vantaggi vi chiederete per quale ragione si continuano ad utilizzare i microprocessori; la ragione è semplice, per poter raggiungere assorbimenti così ridotti e le dimensioni di una lenticchia, la potenza di calcolo deve essere sacrificata.

La parola sacrificata deve essere letta con il corretto peso, infatti non significa che il microcontrollore è destinato ad essere utilizzato in applicazioni tipo il pallottoliere, mentre il microprocessore farà calcoli astrofisici. Infatti la tecnologia e la potenza di calcolo dei computer a bordo dell'Apollo 11, per mandare l'uomo sulla luna, era inferiore a quella attualmente possibile da alcuni microcontrollori. In ultimo è giusto citare il fatto che alcuni microcontrollori altro non sono che vecchi progetti di microprocessori estesi a diventare microcontrollori. Per esempio è facile trovare microcontrollori ad 8 bit basati sulla CPU Z80 o sull'8051. In particolare il core 8051 viene frequentemente utilizzato come CPU all'interno di circuiti integrati più complessi. Infatti, quando bisogna aggiungere delle capacità di calcolo a degli integrati, anche se non sono microcontrollori, è più facile inserire un core (modulo)

<sup>1</sup> Poiché il microcontrollore possiede al suo interno un gran numero di sistemi, viene anche detto SoC, ovvero System on Chip.

---

bello e pronto che non reinventare l'acqua calda (o la ruota, come dicono i popoli anglosassoni). Infatti far uso di un core significa includere un file di libreria e far uso dei nomi relativi agli input e gli output del core. Da quanto detto, non escludo che qualche microprocessore attuale diventerà un “banale” core per microcontrollori<sup>2</sup> del futuro.

## I sistemi embedded

Capiti i pregi e difetti del microprocessore e del microcontrollore vediamo di descrivere brevemente le applicazioni dove i microcontrollori vengono utilizzati. Esempi tipici sono elettrodomestici, giocattoli, sistemi di misura dedicati e applicazioni consumer in generale. Tutte queste applicazioni rientrano in quella più ampia nota come applicazione embedded. Le applicazioni embedded sono un gruppo di applicazioni che generalmente viene descritta come un'applicazione mirata per un determinato scopo e il cui hardware (risorse) disponibili sono ridotte al minimo. Per tale ragione applicazioni con i microcontrollori rientrano nella categoria delle applicazioni embedded. Nonostante un'applicazione embedded sia ritenuta un'applicazione con ridotte risorse hardware, tale descrizione è un po' vaga. Infatti molte schede di sviluppo utilizzate in applicazioni embedded altro non sono che mini schede di PC il cui microprocessore era il top qualche anno prima. Per esempio schede per applicazioni embedded fanno frequentemente uso di microprocessori Intel della serie Pentium II® e Atom™. Il Pentium II® quando venne introdotto sul mercato era un gioiello della tecnologia il cui costo portava il prezzo di un PC pari al valore di uno stipendio medio italiano del tempo. Oggi è possibile acquistare schede embedded con Pentium II® a meno di 100 euro.

Da quando detto si capisce dunque che la tipologia di hardware utilizzato in applicazioni embedded varia con il tempo, e quello che oggi è il top della tecnologia, domani sarà utilizzato per applicazioni embedded. In tutto questo, è però possibile dire che per definizione se fai utilizzo di un microcontrollore invece di un microprocessore, a buon diritto il sistema è da considerarsi embedded. In caso contrario sarà il tempo a decidere.

In ultimo vediamo brevemente cosa significa programmare per un sistema embedded. Qualora si faccia utilizzo di un microcontrollore di fascia bassa, ovvero 8 o 16 bit, frequentemente il software, escluso le librerie utilizzabili ed offerte frequentemente con il compilatore, viene scritto dal nulla e in maniera specifica per l'applicazione. Negli ultimi anni sono comparsi dei piccoli sistemi operativi RTOS (Real Time Operative System) dedicati ad applicazioni generiche e utilizzabili su vari microcontrollori in cui sia richiesta la funzione Real Time. Nel caso in cui si stia sviluppando un sistema utilizzando una piattaforma con microcontrollore o microprocessore a 32 bit, la questione dello sviluppo software è in generale differente. Per queste strutture, qualora la CPU o MCU possieda la MMU (Memory Managment Unit), è possibile utilizzare sistemi operativi standard quali Linux<sup>3</sup> e Windows.

L'installazione e la gestione di applicazioni naturalmente non è semplice come per un PC, visto che i sistemi operativi si trovano ad essere installati in ambienti un po' “stretti”! Alcuni esempi di applicazioni a 32 bit dove vengono spesso utilizzati sistemi operativi come Linux e Windows, sono i palmari, cellulari, router, stampanti e molti altri. Famoso fu un esperimento in cui venne installato Linux su un orologio da polso...dunque le applicazioni embedded sono intorno a noi!

---

<sup>2</sup> Come si vedrà a breve il core ARM è spesso utilizzato per microcontrollori.

<sup>3</sup> Vi sono alcuni progetti in cui vengono supportati anche CPU e MCU a 16 bit e senza MMU, ma il loro supporto e potenzialità è ridotto. Per tale ragione, qualora si voglia far uso di Linux, è consigliabile utilizzare strutture a 32 bit con MMU.

## I microcontrollori della Microchip

Microchip produce una vasta gamma di microcontrollori per applicazioni generiche (MCU) e microcontrollori per applicazione mirate all'elaborazione digitale di segnali analogici (DSC, Digital Signal Controller). La scelta spazia da microcontrollori a 8, 16 e 32 bit. La famiglia ad 8 bit è quella che fa di Microchip la società leader dei microcontrollori con più di 250 tipi, mentre le strutture a 16 e 32 bit, come vedremo, hanno molti più competitori. Cominciamo per gradi e descriviamo le caratteristiche offerte dalle varie famiglie 8, 16, 32 bit.

- **8 bit**

Come detto la famiglia ad 8 bit rappresenta la più vasta, questa si divide in:

- × Architettura base (Base Architecture)
- × Architettura media (Midrange Architecture)
- × Architettura media potenziata (Enhanced Midrange Architecture)
- × Architettura PIC18 (PIC18 Architecture)

In Tabella 1 sono riportate le caratteristiche principali di ogni famiglia con i relativi microcontrollori che ne fanno parte.

	Architettura base	Architettura media	Architettura media pot.	Architettura PIC18
<b>Numero pin</b>	6-40	8-64	8-64	18-100
<b>Performance</b>	5MIPS	5MIPS	8MIPS	10MIPS-16MIPS
<b>Memoria programma</b>	Fino a 3KB	Fino a 14KB	Fino a 56KB	Fino a 128KB
<b>Memoria Dati</b>	Fino a 138B	Fino a 368B	Fino a 4K	Fino a 4K
<b>Interruzioni</b>	no	Si (singola)	Si (singola)	Si (multipla)
<b>PIC</b>	Include PIC10, PIC12 e PIC16	Include PIC12 e PIC16	Include PIC12F1 e PIC16F1	PIC18

**Tabella 1:** Caratteristiche principali delle varie famiglie di microcontrollori a 8 bit

Come da Tabella 1 è possibile vedere che Microchip realizza microcontrollori con un numero di pin a partire da soli 6 pin. Questo realmente significa che è possibile inserire della piccola intelligenza a piccoli dispositivi elettronici. La famiglia più performante tra quelle ad 8 bit è proprio la famiglia PIC18, di cui andremo a descrivere l'utilizzo del compilatore C.

- **16 bit**

La famiglia a 16 bit si suddivide in microcontrollori a 16 bit e DSP (Digital Signal Processor) noti come dsPIC. Di questa fanno parte i PIC24 e i dsPIC30, dsPIC33. I dsPIC si differenziano dalla serie PIC24 solamente per la presenza, nella serie dsPIC del core DSP. L'aggiunta del core DSP fa dei dsPIC la scelta ideale in applicazioni audio, controllo motore e grafiche (con risoluzione QVGA). Nella famiglia 16 bit è possibile trovare periferiche tipo ADC 10/12 bit, DAC 10/16 bit, USB, DMA (Direct Memory Access<sup>4</sup>). La potenza di calcolo dei PIC a 16 bit è di 40 MIPS.

<sup>4</sup> La presenza del DMA permette di gestire un grosso flusso dati da diverse periferiche interne e la memoria dati, senza l'intervento della CPU. Per esempio se si volesse fare una FFT con risoluzione a 64 campioni, sarebbe possibile

---

- **32 bit**

La famiglia a 32 bit è stata recentemente introdotta da Microchip. Con un certo ritardo rispetto ai grandi produttori di microcontrollori i PIC32 hanno cominciato ad essere utilizzati nel mercato. L'architettura dei PIC32, diversamente da molti microcontrollori a 32 bit non è basata su architettura ARM. I microcontrollori a 32 bit hanno frequenze di clock fino a 80MHz, il che significa che non può essere confrontato, come si vedrà a breve, con alcuni DSP a 300MHz le cui applicazioni sono naturalmente differenti. Ciononostante tra i DSP nella stessa fascia di frequenza di clock, la struttura utilizzata da Microchip non ha nulla da invidiare a quelli basate su architetture ARM.

Unico neo dei PIC32 è che sono arrivati tardi sul mercato. Questo ritardo non ha nulla a che fare con le performance attuali, ma ha ridotto Microchip in una più ristretta fascia di mercato di partenza.

## Cosa c'è oltre al mondo Microchip

Dopo tutto questo parlare di Microchip rischio d'indottrinarvi a tal punto che ignorerete il mondo esterno. Il mondo è bello perché è vario, e il mondo dei microcontrollori e delle logiche programmabili non è da meno. Far utilizzo di soluzioni di un solo costruttore può essere limitativo, anche se certamente pratico. Apriamo la finestra e vediamo cosa possiamo trovare:

### FPGA e CPLD

Come prima cosa, è doveroso almeno citare le logiche programmabili per eccellenza, le CPLD e FPGA<sup>5</sup>. CPLD è l'acronimo inglese “Complex Programmable Logic Device” mentre FPGA è l'acronimo inglese di “Field Programmable Gate Array”. Tale dispositivi sono composti rispettivamente da migliaia e centinaia di migliaia di porte elementari o gate. La differenza principale tra le due famiglie di dispositivi è proprio il numero di porte e le tipologie di moduli integrati, ovvero la complessità interna, quindi il costo! La programmazione consiste nel bruciare o meno alcuni fusibili<sup>6</sup> per permettere o meno la connessione di determinate porte. In questo modo è possibile creare anche piccole CPU<sup>7</sup> o comunque una qualunque funzione logica. Frequentemente le FPGA e CPLD già incorporano moduli elementari quali CPU o ALU<sup>8</sup>. Questi dispositivi programmabili vengono utilizzati al posto dei microcontrollori quando è richiesto un maggior “data throughput” ovvero la quantità di dati che passerà al loro interno. Per comunicazioni ad alta velocità, gestione bus o macchine a stati, sono in generale da preferirsi. In particolare le FPGA più potenti possono essere utilizzate in applicazioni dove le CPU più veloci risultano troppo lente! Non è un caso che applicazioni militari tipo missili abbondano di FPGA!

Visti i costi limitati dei CPLD, questi risultano da un punto di vista economico anche una valida alternativa ai microcontrollori, ma come sempre il tutto dipenderà dallo specifico progetto, quindi dalle sue specifiche.

---

mandare i campioni dall'ADC direttamente alla memoria dati, e far sì che la CPU inizi ad accedere ai campioni, solo quando sono presenti tutti e 64.

<sup>5</sup> I loro predecessori, ancora utilizzati ma più per esigenze di vecchi progetti che per nuovi, sono i PLD e PAL. Il CPLD è praticamente uguale ma supporta un numero maggiore di gate e moduli interni. Tra i più grandi produttori di FPGA e CPLD si ricorda Xilinx e Altera.

<sup>6</sup> I fusibili dei primi modelli di FPGA e CPLD permettevano una sola scrittura. Le tecnologie attuali permettono invece di riscrivere i dispositivi come se fossero delle memorie Flash.

<sup>7</sup> CPU è l'acronimo inglese di “Central Processing Unit”.

<sup>8</sup> ALU è l'acronimo inglese per “Arithmetic Logic Unit”.

---

## DSP

I DSP, acronimo inglese di “Digital Signal Processing, o Processor”, sono praticamente dei microcontrollori ottimizzati per applicazioni di elaborazione dei segnali. Normalmente quello che li differenzia da microcontrollori standard non è tanto la frequenza di clock, che comunque è più o meno elevata<sup>9</sup>, bensì la presenza di più accumulatori (registri speciali per fare operazioni), operatori di somma e spostamento ottimizzati per implementare filtri FIR<sup>10</sup> e IIR<sup>11</sup>. I DSP sono normalmente utilizzati per applicazioni audio e video. Molti cellulari e ripetitori per cellulari, oltre ad avere FPGA hanno frequentemente anche DSP. Per tale ragione le FPGA più potenti stanno avendo la tendenza d'implementare core (moduli) DSP direttamente all'interno dell'FPGA.

## ARM

ARM, acronimo di “Advanced RISC Machine”, rappresenta un core venduto su licenza dalla ARM Holding Ltd. ed integrato all'interno di microprocessori e microcontrollori. Come vedremo nei Capitoli successivi i PIC della microchip utilizzano una struttura RISC non di tipo ARM. L'utilità di far uso di microprocessori o microcontrollori basati sul core ARM sta nel fatto che sono tra loro più o meno compatibili e vi sono strumenti di sviluppo e Debug prodotti da molte società. Questa varietà e ricchezza di strumenti sta permettendo una continua espansione del core ARM di cui vengono rilasciate continuamente nuove versioni. Un'altra ragione per cui il core ARM sta avendo successo è legato al fatto che in applicazioni embedded viene sempre più utilizzato Linux. Questo può essere compilato per mezzo di cross compiler anche per il core ARM, ponendo questo core al top delle applicazioni Linux embedded. La struttura ARM non è però la sola a condividere le grazie di Linux, microprocessori a 32 bits basati su 80x86 e MIPS sono altrettanto usati.

## Altri microcontrollori

Un po' poco gentile chiamare un sotto paragrafo altri microcontrollori ma sono talmente tanti e ognuno meriterebbe un testo a parte, così sono costretto a riassumere il tutto. Faccio presente che questo aggregamento non li deve affatto mettere in ombra visto che i microcontrollori dei produttori descritti sono molto utilizzati e Microchip stessa è in continua competizione.

- **ATMEL**

Senza dubbio il competitore con il quale Microchip si trova a dividere il mercato dei microcontrollori ad 8 bit. Per tale ragione nella crisi economica globale del 2009 ha cercato di comprarla. Atmel produce i microcontrollori ad 8 bit della famiglia AVR®. Allo stesso modo di TI, Atmel ha sviluppato microcontrollori a 32 bit AVR32® basati su una struttura RISC proprietaria, ma allo stesso tempo ha prodotto anche microcontrollori a 32 bit basati su core ARM (AT91SAM®). Oltre a questo, Atmel produce microcontrollori con core basato sul microprocessore 8051. Uno dei punti di forza dei microcontrollori 32 bit basati su core ARM è legato al fatto che sono supportati dalla comunità di Linux. In particolare i microcontrollori ARM sono forse secondi solo alla struttura 80x86, ARM e MIPS.

---

<sup>9</sup> I DSP più potenti attualmente sul mercato hanno un dual core e frequenza di clock di 2GHz.

<sup>10</sup> FIR è l'acronimo inglese per “Finite Impulse Response”.

<sup>11</sup> IIR è l'acronimo inglese per “Infinite Impulse Response”.

---

- **ST**

La ST Microelectronics è un altro grosso competitore produttore di microcontrollori ad 8 bits. Famosa è la famiglia ST6® e ST7® . Oltre ai microcontrollori ad 8 bit la ST produce microcontrollori a 16 e 32 bit. Quelli a 16 bit sono la famiglia ST10 mentre quelli a 32 bit sono basati su core ARM.

- **TI**

La Texas Instrument è uno dei colossi del mondo dell'elettronica, praticamente produce tutti i tipi di componenti elettronici, eccetto quelli discreti. Dal punto di vista dei microcontrollori produce microcontrollori a 16 e 32 bit. La famiglia a 16 bit è nominata MSP430 mentre per la famiglia a 32 bit è la C2000. A 32 bit la TI offre anche microcontrollori basati sul core ARM. Le potenze di calcolo dei microcontrollori TI, grazie alla loro elevata frequenza di calcolo, non hanno molti rivali. Oltre ai microcontrollori la TI produce diverse famiglie di DSP

- **Analog Devices**

L'Analog Devices non è un vero competitore della Microchip, visto che si è differenziata producendo un ristretto numero di microcontrollori ottimizzati per applicazioni analogiche. In particolare possiede microcontrollori di fascia alta basati come per TI sul core ARM. L'Analog Devices ha microcontrollori sia a 16 che 32 bit. Oltre ai microcontrollori di fascia alta, possiede qualche microcontrollore basato sul core 8052.

- **Maxim**

Maxim IC, utilizza un approccio simile a quello dell'Analog Devices, ovvero produce microcontrollori a 16 bit per applicazioni specifiche. La scelta dei microcontrollori a disposizione non è paragonabile con quella della Microchip. Il core principale dei microcontrollori della Maxim è il core 8051.

- **Zilog**

La Zilog è particolarmente famosa per aver introdotto il primo microprocessore di successo, lo Z80. Attualmente lo studio dello Z80 è previsto nei piani ministeriali degli istituti tecnici industriali. Nonostante il successo dello Z80 lo stato attuale non è come in passato, ma senza dubbio produce microcontrollori di qualità. In particolare produce microcontrollori ad 8,16 e 32 bit.

- **NXP**

NXP nata dalla divisione componenti elettronici della Philips, attualmente produce una vasta gamma di microcontrollori. In particolare possiede microcontrollori ad 8 bit basati sul core 8051 e un ridotto numero di microcontrollori a 16 bit. Dall'altro lato possiede una vasta scelta di microcontrollori a 32 bit basati sul core ARM.

- **Renesas**

La Renesas è una delle società leader nella produzione di microcontrollori e microprocessori. Possiede un'ampia scelta di microcontrollori che va dagli 8 bit fino ai 32 bit. Particolarmente famosi sono i SuperH a 32 bit, supportati dai cross compiler Linux, dunque i SuperH supportano Linux embedded.

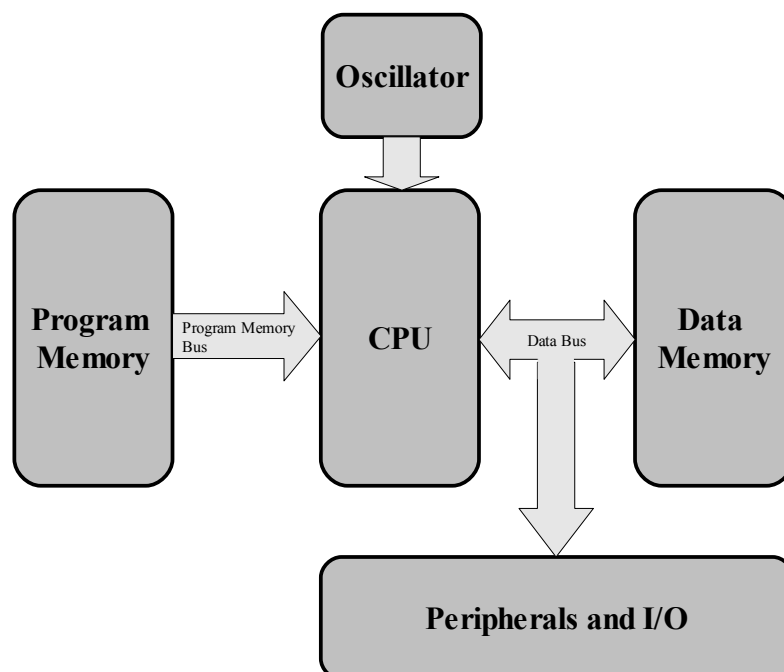
# Capitolo II

## Architettura ed Hardware dei PIC18

In questo Capitolo verrà descritta l'architettura base dei PIC18, ma dal momento che nel testo si farà uso del PIC18F4550, maggiori dettagli verranno dati per tale PIC. Il Capitolo non vuole sostituire il datasheet, al quale si rimanda per tutti i dettagli tecnici. Nel Capitolo non verranno affrontate tutte le periferiche che è possibile trovare all'interno dei PIC essenzialmente per due ragioni: la prima è legata al fatto che molte periferiche hardware non sono presenti in tutti i PIC, dunque nei casi specifici si rimanda al datasheet del PIC utilizzato. Una seconda ragione è legata al fatto che alcune delle periferiche interne sono descritte in dettaglio nei Capitoli successivi, dove vengono riportati i programmi di esempio per il loro utilizzo. Questo permetterà d'imparare la periferica direttamente con degli esempi tra le mani.

### Architettura dei PIC18

I PIC18 come anche le altre famiglie di PIC hanno un'architettura di tipo Harvard ed una CPU (Central Processing Unit) di tipo RISC (Reduced Instruction Set Computer).



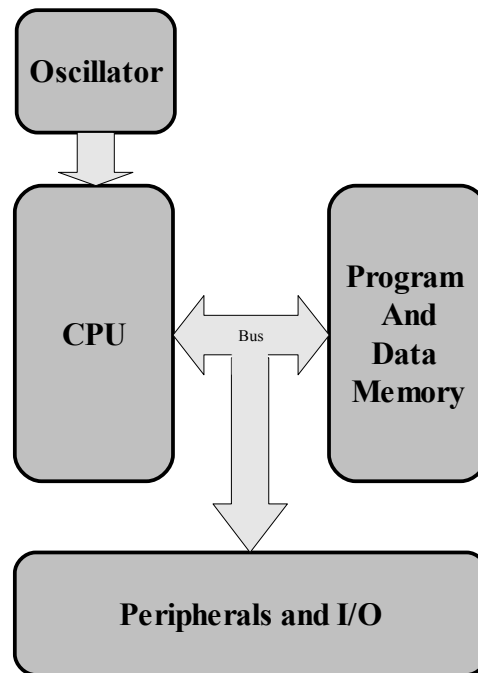
**Figura 1:** Architettura base dei PIC18 (Harvard)

La struttura Harvard, riportata in Figura 1, è caratterizzata dal fatto che la memoria utilizzata per il programma (Program Memory) e quella utilizzata per i dati (Data Memory)



possiedono un bus dedicato. Questo significa che non sono presenti conflitti legati al contemporaneo utilizzo della memoria, per la lettura di dati o del programma. Questo accorgimento, se da un lato complica la struttura, dall'altro aumenta la flessibilità nella gestione della memoria. Inoltre, come si vedrà a breve, la memoria dati e programma sono comunque differenti dunque la struttura Harvard rappresenta una buona scelta per la struttura intrinseca di un microcontrollore.

In contrapposizione alla struttura Harvard è presente la struttura di von Neumann, la quale possiede un unico bus per la lettura della memoria condivisa tra programma e dati. In Figura 2 è riportato lo schema a blocchi di una struttura di von Neumann.



**Figura 2:** Architettura di “von Neumann”

Tale struttura, è quella utilizzata nei computer classici, infatti i microprocessori hanno in genere un unico bus con cui possono accedere la memoria RAM, nella quale sono presenti sia i programmi che i dati. I microprocessori di ultima generazione non hanno nemmeno accesso diretto alla RAM, la quale viene gestita per mezzo dell'unità DMA (Direct Memory Access).

Per mezzo di cache di vario livello e tecniche di predizione dei salti, si cerca di avere sempre in cache quello di cui si ha bisogno. La cache è una delle tecniche per mezzo delle quali si ha un accesso veloce ai dati e al programma.

Oltre all'architettura Harvard, si è detto che la CPU è di tipo RISC. Il nome RISC discende dal fatto che il numero di istruzioni eseguibili dal PIC sono limitate se paragonate a quelle messe a disposizione da un microprocessore classico con CPU di tipo CISC (Complex Instruction Set Computer). Infatti i PIC18 possiedono 75 istruzioni base<sup>12</sup> mentre microprocessori tipo il Pentium ne possiedono diverse centinaia. Come per il discorso effettuato per i sistemi embedded, parlare di RISC e CISC semplicemente su una base numerica non ben definita, fa intendere che in futuro cioè che era considerato CISC verrà forse considerato RISC. I PIC della famiglia Midrange (esempio i PIC16) possiedono per esempio 35 istruzioni.

<sup>12</sup> I PIC18 possiedono la modalità estesa che aggiunge 8 ulteriori istruzioni a quelle base. Tale istruzioni sono state storicamente aggiunte per poter ottimizzare la compilazione del codice C.



Una breve nota meritano gli altri famosi microcontrollori di tipo RISC attualmente presenti sul mercato, tra questi si ricordano: ARM, AVR, MIPS e POWER.

Dopo Questa breve nota introduttiva sull'architettura dei PIC18 vediamo qualche dettaglio in più, in particolare considerando la struttura interna del PIC18F4550, riportata in Figura 3.

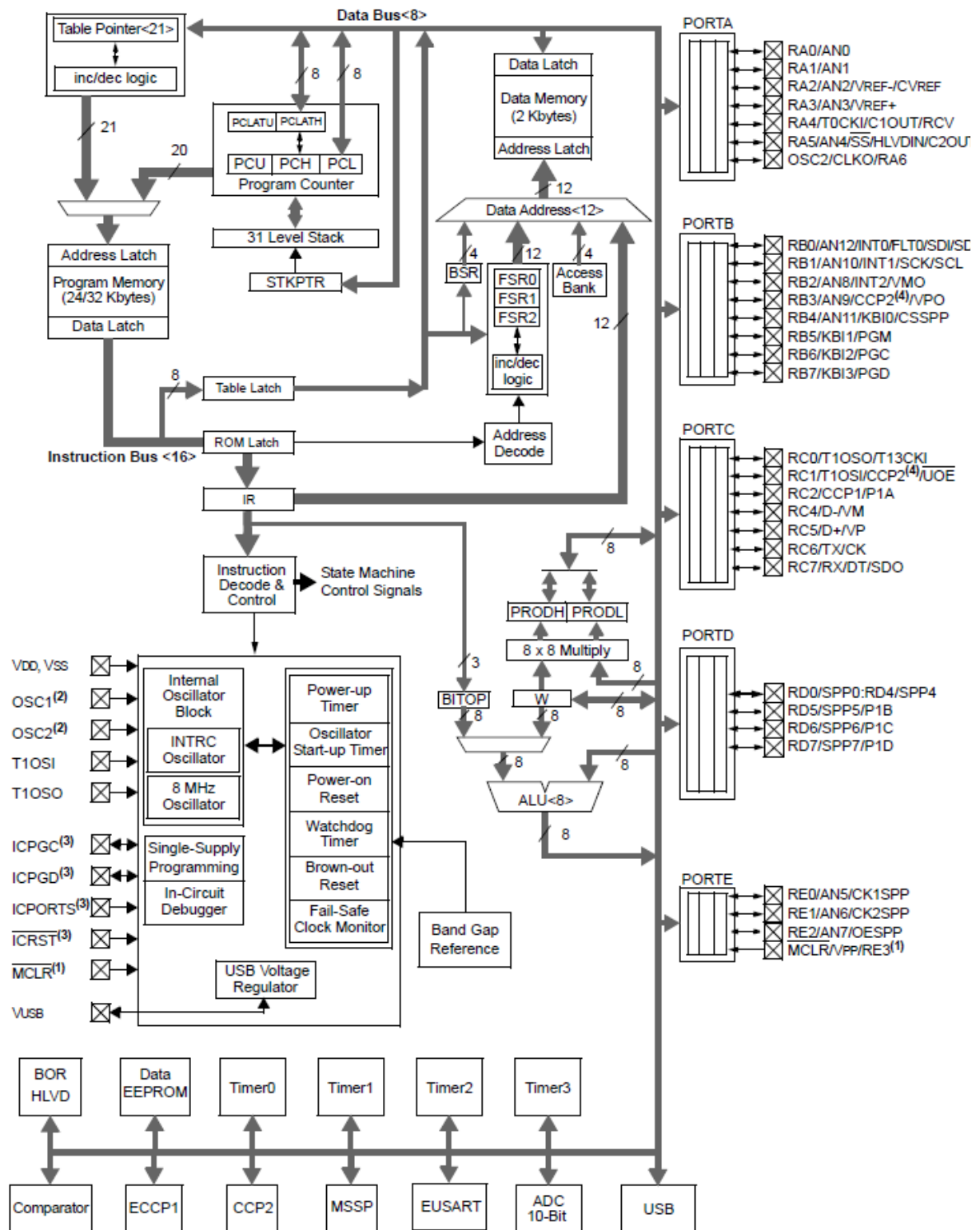


Figura 3: Architettura del PIC18F4550

---

Dallo schema a blocchi, presente in ogni datasheet è immediato vedere le periferiche che possiede il PIC d'interesse ed in particolare l'organizzazione dei registri, della memoria e delle periferiche. Vengono inoltre messi in evidenza i vari bus che interconnettono le varie periferiche e il loro parallelismo, ovvero il numero di linee disponibili. In particolare è possibile vedere che la memoria programma e dati sono separate. Una caratteristica importante introdotta nei PIC18 è la presenza del moltiplicatore 8x8, per mezzo del quale è possibile ridurre notevolmente il numero di istruzioni (quindi il tempo di esecuzione) necessario per l'esecuzione di moltiplicazioni. Un'altra importante caratteristica presente nei PIC18, non visibile a livello di schema a blocchi, è l'introduzione dei livelli di priorità.

Diversamente dalla famiglia Midrange, i PIC 18 hanno infatti un livello di priorità delle interruzioni alto ed uno basso. Il livello d'interruzione alto è compatibile con quello presente anche nei PIC16. E' importante far notare che i quadrati con la croce interna rappresentano i pin fisici del PIC. Si osservi che sulla destra della Figura 3 sono presenti diversi blocchi nominati PORTx. Tali blocchi rappresentano le unità principali del PIC per mezzo dei quali è possibile creare dei percorsi di input o di output dal PIC stesso.

---

## Organizzazione della Memoria

Ogni microcontrollore possiede al suo interno almeno due tipi di memoria, quella per memorizzare il programma in maniera permanente e la memoria dati. Come visto i PIC18 hanno una struttura Harvard, ovvero la memoria programma e dati hanno bus dedicati, permettendo il loro simultaneo accesso. I PIC18 come anche i PIC delle altre famiglie possiedono in particolare i seguenti tipi di memoria:

- Memoria Programma (Program Memory)
- Memoria Dati (Data Memory)
- Memoria EEPROM (EEPROM Memory)

Vediamo ora in maggior dettaglio i vari tipi di memoria e le loro caratteristiche.

### Memoria Programma

La memoria programma è quella memoria dove risiede in maniera permanente il programma che il microcontrollore dovrà eseguire. In particolare i PIC18 possiedono una memoria flash<sup>13</sup>. Tale tipo di memoria ha la caratteristica di poter essere cancellata e riscritta più volte (circa 100.000). Un'altra caratteristica importante della memoria flash è quella di poter mantenere il programma memorizzato anche quando il PIC non è alimentato<sup>14</sup>.

Normalmente il programma viene caricato all'interno della memoria per mezzo del programmatore, sfruttando i pin del microcontrollore dedicati alla programmazione. Oltre a questa modalità di programmazione, i PIC18, come anche molti altri PIC di altre famiglie hanno la caratteristica di auto-programmazione per mezzo del software (Self-programming under software control). Questo significa che è possibile scrivere un programma da caricare sul PIC e permettere al programma stesso di utilizzare la memoria flash. Tale caratteristica permette ad un programma di aggiornarsi senza aver bisogno di un programmatore. Questa tecnica è possibile se all'interno del PIC viene caricato un bootloader. La Microchip fornisce vari tipi di bootloader che possono essere integrati con un qualunque programma e permettono il loro aggiornamento per mezzo della porta seriale RS232, USB e CAN. E' comunque possibile utilizzare un qualunque altro canale di comunicazione per mezzo del quale può essere trasferita la nuova versione di programma che si vuole installare.

La caratteristica di auto-programmazione può essere utilizzata per memorizzare in maniera permanente dei dati evitando di utilizzare la EEPROM, di dimensioni più ridotte. Ciononostante l'utilizzo della memoria flash per memorizzare dati può essere più complicata della memoria EEPROM, che può essere letta e scritta per singole celle di memoria. Infatti la memoria flash dei PIC18 può essere scritta e cancellata solo per blocchi<sup>15</sup>. In particolare la memoria può essere letta un byte alla volta, scritta 32 byte alla volta e cancellata 64 byte alla volta<sup>16</sup>. Questa divisione in blocchi può portare ad un notevole spreco di memoria o lavoro di salvataggio e riscrittura dati.

Dopo questa panoramica sulle caratteristiche fisiche della memoria programma, vediamo

---

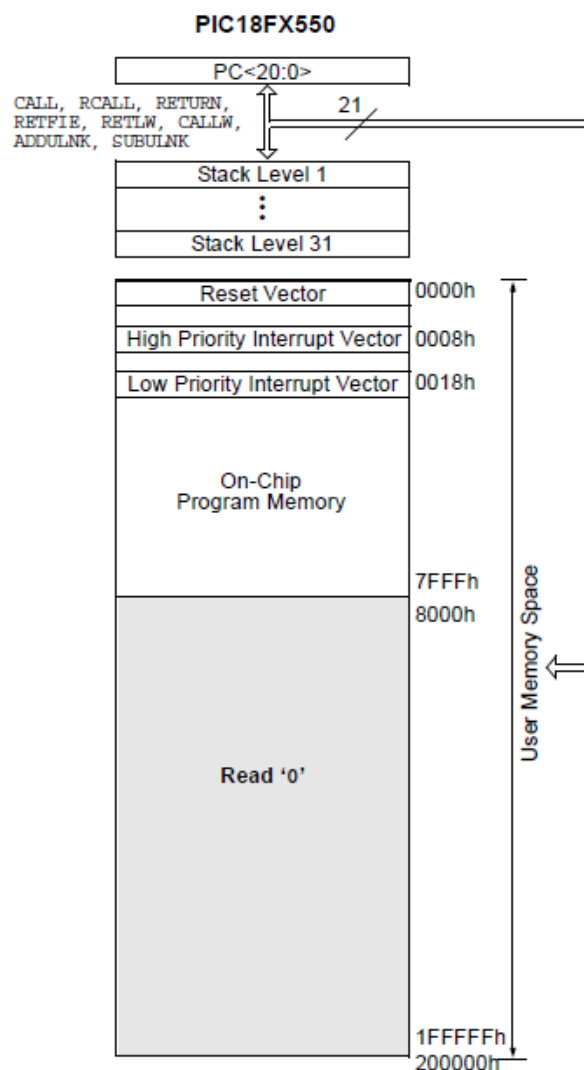
<sup>13</sup> E' importante ricordare che vi sono anche PIC di altre famiglie, che al posto della memoria flash, fanno uso di una memoria OTP, ovvero scrivibile una sola volta.

<sup>14</sup> La memoria flash, come anche quella EEPROM, sono specificate per mantenere memoria dei dati per oltre 40 anni; naturalmente tale valore è solo un valore statistico.

<sup>15</sup> La necessità di lettura, scrittura e cancellazione in blocchi è una caratteristica delle memorie flash, non legata alla struttura del PIC18.

<sup>16</sup> Prima di poter scrivere nuovi dati su di un blocco di memoria, è necessario effettuare la cancellazione dei vecchi dati.

in maggior dettaglio le sue caratteristiche ed utilizzo da parte della CPU. Abbiamo per ora assodato che il programma che andremo a scrivere in C, una volta convertito in linguaggio macchina, verrà caricato in memoria, è importante ora capire dove verrà caricato. La memoria programma è una memoria flash la cui larghezza della parola memorizzata per cella è di un byte, ovvero 8 bit. Le istruzioni del PIC sono per la maggior parte di ampiezza pari a due byte, ma ce ne sono alcune (CALL, MOVFF, GOTO, e LSFR) che necessitano di 4 byte. Questo significa che per ogni istruzione saranno necessarie rispettivamente 2 o 4 celle di memoria. In Figura 3 è possibile vedere che la memoria programma viene letta facendo uso del registro speciale PC (Program Counter). Tale registro è di 21 bit, ovvero potenzialmente permette di puntare fino a 2MByte<sup>17</sup> di memoria. Il valore del registro PC rappresenta l'indirizzo di memoria programma associata all'istruzione che deve essere caricata (fetch). Quando l'istruzione viene caricata viene incrementato di 2 in modo da puntare l'istruzione successiva<sup>18</sup>. Il PIC18F4550 possiede 32KByte di memoria, ovvero è possibile scrivere circa 16.000 istruzioni. Se il PC viene impostato ad un valore di memoria non valido, il valore letto è pari a 0 come visibile in Figura 4.



**Figura 4:** Architettura della memoria programma del PIC18F4550

<sup>17</sup> Attualmente non vi sono PIC18 che possiedono 2MByte di memoria.

<sup>18</sup> Nel caso di istruzioni a 4 byte, il registro PC viene incrementato di 4.

---

La Figura 4 mette in evidenza il fatto che la memoria programma è sequenziale, dal valore 0000h (valore espresso in esadecimale) al suo valore massimo. Dalla figura è possibile però leggere che l'indirizzo 0h è nominato Reset Vector, mentre l'indirizzo 08h e 18h sono rispettivamente nominati High Priority Interrupt Vector e Low Priority Interrupt Vector. Questi indirizzi di memoria hanno un significato speciale e deve essere ben compreso. In particolare scrivendo in C ci si scorderà quasi del Reset Vector e di quanto si dirà a breve, mentre quanto verrà detto per gli Interrupt Vector verrà richiesto anche durante la programmazione in C e verrà ripresa a tempo debito.

L'indirizzo 0h, ovvero Reset Vector, rappresenta l'indirizzo a cui viene posto il PC quando viene resettato il microcontrollore. Questo rappresenta dunque il punto di partenza del nostro programma, ed è infatti in questo punto che il programmatore inserirà le nostre prime istruzioni<sup>19</sup>. A questo punto vi sarete resi conto che il secondo indirizzo speciale è posto a soli 8 indirizzi dopo, dunque il nostro programma non può essere memorizzato in maniera sequenziale. Infatti quello che il compilatore fa, o quello che il programmatore dovrebbe fare scrivendo il programma in assembler, è quello di utilizzare gli indirizzi iniziali solo per memorizzare un salto, che normalmente va a puntare la funzione main del nostro programma. A partire dal punto in cui si trova la funzione main, o quale che sia la funzione, il programma verrà scritto in maniera più o meno sequenziale. La non sequenzialità discende dal fatto che un programma è in generale composto da più funzioni, il che si traduce in blocchi d'istruzione separati. In ogni modo programmando in C ci si può scordare di questi dettagli.

Gli indirizzi 08h e 18h rappresentano gli indirizzi particolari al quale viene posto il registro PC quando viene generata un'interruzione. In particolare un'interruzione ad alta priorità porrà il PC al valore 08h mentre un'interruzione a bassa priorità porrà il PC al valore 18h. Come per il caso del Reset Vector, il numero di locazioni disponibili tra i due Interrupt Vector non è sufficiente per gestire eventuali periferiche che hanno generato un'interruzione, per tale ragione le locazioni di memoria associate agli Interrupt Vector sono in generale utilizzate per fare un salto alla funzione che gestirà le interruzioni. Maggiori dettagli verranno mostrati nel paragrafo dedicato alle interruzioni.

Oltre al registro PC, in Figura 4 è possibile vedere che in alto sono presenti delle celle di memoria particolari nominate Stack. In particolare sono presenti 31 celle di memoria della larghezza di 21 bit (giusti giusti per memorizzare il registro PC). Tali locazioni di memoria rappresentano un buffer di indirizzi nel quale viene memorizzato il registro PC ogni volta che bisogna fare un salto nel nostro programma, sia per eseguire una funzione generica sia per eseguire una routine associata all'interruzione. Questo buffer è necessario poiché quando si interrompe il normale flusso del programma è necessario memorizzare l'indirizzo al quale bisognerà tornare (ovvero quale istruzione eseguire) al termine dell'esecuzione delle istruzioni associate alla funzione che ha interrotto il flusso normale. Il valore del PC che viene salvato è proprio quello dell'istruzione successiva che sarebbe stata eseguita. Gli indirizzi del PC vengono salvati nel buffer in pila, ovvero l'ultimo indirizzo salvato sarà il primo al quale ritornare, e così via. Dal momento che sono presenti 31 livelli di Stack, sarà possibile utilizzare fino a 31 livelli annidati di funzioni<sup>20</sup>. Allo Stack è associato un registro particolare nominato Stack Pointer (STKPTR), che ha il compito di puntare il livello di ritorno, ovvero indirizzo, che deve essere caricato nel registro PC.

In ultimo è bene ricordare che sia il Program Counter che lo Stack Pointer possono essere gestiti manualmente via software facendo uso di registri interni dedicati. Le modalità della

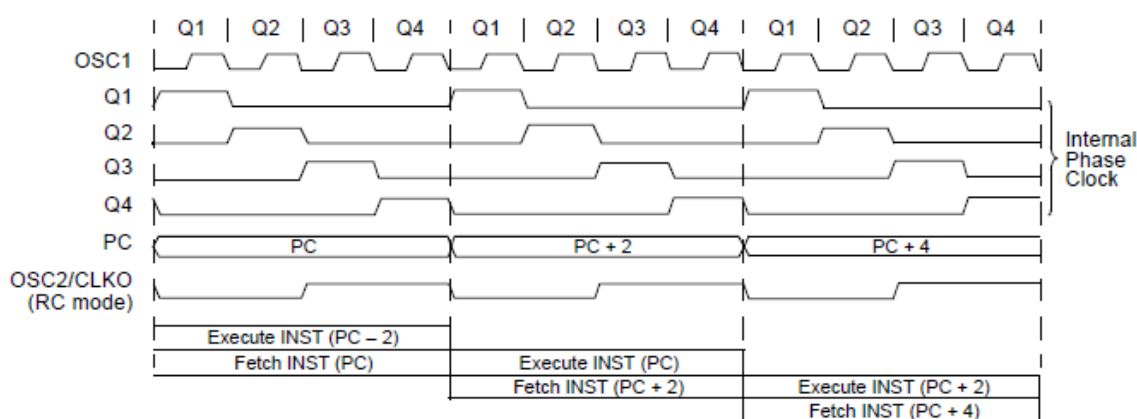
---

<sup>19</sup> Il Reset Vector è presente in tutti i microcontrollori e Microprocessori di qualunque marca. L'indirizzo a cui questo viene posto può differire a seconda dei modelli.

<sup>20</sup> Nel caso si faccia uso della modalità DEBUG, 5 livelli saranno utilizzati dalla funzionalità di DEBUG, dunque devono essere preventivamente considerati per evitare problemi.

loro gestione e applicazioni esulano dagli scopi di questo testo. La loro gestione è di fondamentale importanza nel caso in cui si faccia uso di sistemi operativi RTOS (Real Time Operative System) dedicati.

Quanto detto fin ora è legato strettamente alla struttura della memoria e dei registri necessari alla sua gestione e controllo. Per far funzionare il tutto è necessario un clock per sincronizzare i vari blocchi e permettere il giusto incremento o decremento dei registri per la gestione della memoria. Il clock interno al PIC come verrà spiegato nel paragrafo dedicato, viene generato per mezzo dell'oscillatore interno, in particolare vi sono diverse opzione che è possibile selezionare. Usando un quarzo esterno da 20MHz, la CPU non lavorerà direttamente a tale frequenza bensì ad un quarto della frequenza principale, facendo uso di altri quattro segnali di clock, tra loro sfasati, ottenuti internamente dividendo per 4 il clock principale; la Figura 5 riporta il clock principale e le sue fasi derivate dalla divisione per 4. In particolare 4 cicli del clock principale rappresentano un ciclo istruzione, ovvero sono necessari 4 cicli del clock principale per poter eseguire un'istruzione interna.



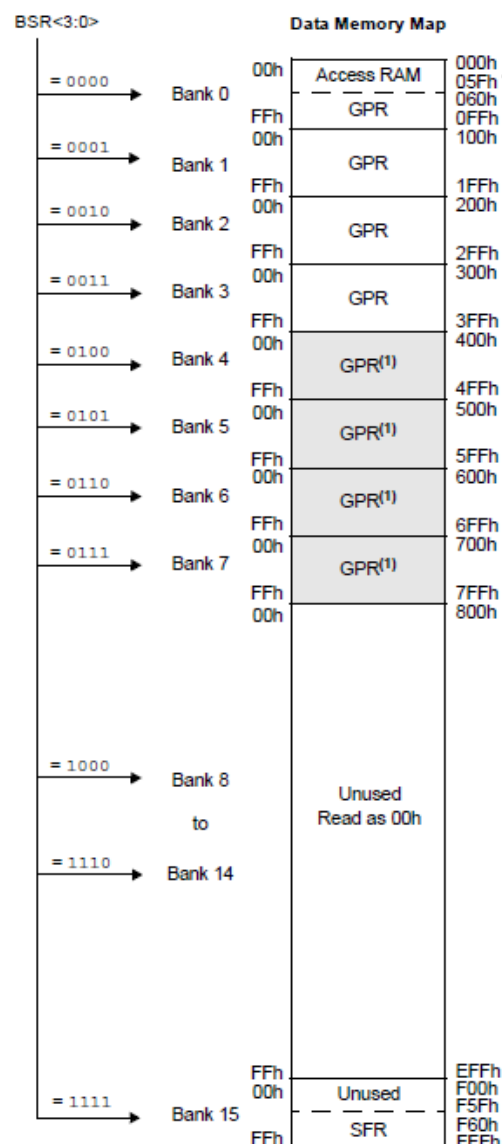
**Figura 5:** Clock principale e clock interno

Quanto detto è visibile nella parte bassa della Figura 5, dove è possibile vedere che durante i 4 cicli di clock che rappresentano un ciclo istruzione, viene eseguita una fase di caricamento istruzione e una decodifica. Bisogna però porre attenzione al fatto che lo schema a blocchi indica che la fase di caricamento dell'istruzione (nominata Fetch) è relativa al valore del registro PC; ovvero c'è la preparazione per caricare l'istruzione presente nella locazione PC e PC+1 la quale verrà caricata nell'Instruction Register (IR) durante la fase di Execute. La fase di esecuzione (nominata Execute) è invece associata al valore del registro PC-2, ovvero all'istruzione precedente, in particolare la fase di esecuzione è preceduta dalla fase di decodifica, ovvero di comprensione dell'istruzione stessa. Le due fasi di Fetch ed Execute sono eseguite per mezzo di una pipeline ovvero sono eseguite in parallelo<sup>21</sup>, dunque anche se si carica un'istruzione e si esegue l'istruzione precedente, a tutti gli effetti si ha l'esecuzione di un'istruzione completa ogni 4 cicli del clock principale ovvero ad ogni ciclo di clock istruzione. A questa regola fanno eccezione le istruzioni di salto, le quali richiedono due cicli istruzioni.

<sup>21</sup> Una pipeline di questo tipo è detta a due livelli. Oltre ad un'aggiunta di ulteriore hardware per eseguire e caricare due istruzioni, la sua realizzazione non comporta molti problemi. CPU più complesse, quali per esempio i microprocessori Intel, AMD hanno molti livelli di pipeline per permettere l'esecuzione contemporanea di più istruzioni. Con molti livelli di pipeline l'esecuzione e la gestione del programma diviene molto più complessa. Per esempio in caso di salti condizionali è necessario cercare di predire se il salto sarà da un lato o dall'altro...se la predizione sarà errata sarà necessario gettare tutti i valori nella pipeline e ricominciare i calcoli. A questo livello di complessità anche i compilatori possono ottimizzare il codice macchina per aiutare la pipeline nell'esecuzione del codice. Si capisce dunque che a parità di frequenza, due CPU possono essere una più veloce dell'altra poiché hanno due pipeline differenti.

## Memoria Dati

Ogni programma durante la sua esecuzione ha in genere bisogno di variabili per mantenere determinati dati o risultati di operazioni. Tali informazioni sono spesso memorizzate nella memoria dati. I PIC18 come anche gli altri microcontrollori, implementano la memoria dati per mezzo di memoria RAM (Random Access Memory) di tipo statica, tale tipo di memoria allo spegnimento dell'alimentazione, perde tutti i dati. La larghezza dell'indirizzo utilizzato per puntare la memoria RAM è di 12 bit. Questo permette di indirizzare fino a 4096 locazioni di memoria. Ciononostante la memoria implementata all'interno del PIC può variare da modello a modello, in particolare il PIC18F4550 possiede al suo interno 2048 byte di memoria RAM. La memoria RAM interna al PIC è suddivisa in blocchi (nominati bank) di 256 byte, dunque il PIC18F4550 possiede 8 banchi di memoria come riportato in Figura 6.



**Figura 6:** Divisione in blocchi della memoria

Nonostante la divisione in blocchi la memoria può anche essere letta come se fosse un solo blocco ma gran parte delle istruzioni implementano l'accesso alla memoria per mezzo dei blocchi. Per tale ragione ogni volta che si deve accedere una variabile è necessario sapere in



---

quale blocco è situato. Molte istruzioni che accedono alla memoria possiedono solo 8 bit per l'indirizzo, che è appunto il numero di bit necessario per puntare le 256 locazioni di memoria all'interno di un blocco. I restanti 4 bit, per determinare il blocco, vengono prelevati dai bit BSR3:BSR0 del registro BSR.

La Figura 6 mette in evidenza che gli 8 banchi di memoria sono nominati GPR ovvero General Purpose Register (registri per uso generico). Oltre ai registri GPR sono presenti anche i registri SFRs ovvero Special Function Registers (registri per uso speciale).

I registri per usi speciali sono tutti quei registri che sono utilizzati per impostare le periferiche interne al PIC stesso. Infatti ogni periferica ha sempre dei registri speciali ad essa dedicati, per mezzo dei quali è possibile impostare il comportamento della periferica stessa. Maggiori dettagli verranno visti quando verranno studiate le varie periferiche interne al PIC.

Un registro speciale un po' particolare che è bene citare è lo Status Register (Registro di Stato), tale registro è praticamente presente in ogni architettura sia essa un microcontrollore che un microprocessore. Lo Status Register contiene lo stato aritmetico dell'ALU (Arithmetic Logic Unit) ovvero di quella parte della CPU che viene utilizzata per svolgere tutte le operazioni binarie. In particolare l'ALU svolge le proprie operazioni per mezzo del registro speciale W, nominato accumulatore<sup>22</sup>. Ogni volta che è necessario svolgere delle operazioni queste devono essere svolte all'interno del registro W o tra W ed un altro registro. Ad ogni operazione il registro di stato informerà con opportuni bit se l'operazione ha generato un risultato nullo, un overflow<sup>23</sup> (trabocco), un numero negativo, ed un riporto.

Si fa notare che la descrizione della memoria dati è stata spiegata per sommi capi poiché gran parte dei dettagli associati ai blocchi possono essere trascurati quando si programma in C. Ciononostante, quando si dovesse avere l'esigenza di ottimizzare l'accesso in lettura e scrittura dati all'interno di strutture dati dedicate a particolari applicazioni, può risultare necessario una maggior comprensione dell'architettura della memoria dati e delle sue modalità di accesso. In particolare si ricorda che la modalità estesa dei PIC, con le sue 8 nuove istruzioni, introduce due nuove modalità di accesso alla memoria dati, oltre alle 4 già esistenti, permettendo di ottimizzare ulteriormente il programma. Alcuni problemi con la memoria dati e la sua divisione in blocchi si può avere anche nel caso in cui bisogna creare delle strutture dati che richiedono più di 256 byte o che comunque vengono dislocate in due o più blocchi di memoria. Ultima nota per quanto riguarda il PIC18F4550, è che i blocchi di memoria da 4 a 7 sono dedicati al buffer del modulo USB, qualora questo risulti attivo. Per maggiori dettagli si rimanda al datasheet del PIC utilizzato.

## Memoria EEPROM

Qualora si abbia la necessità di memorizzare un codice segreto durante l'esecuzione del programma, o semplicemente la temperatura massima e minima della giornata, può ritornare utile l'utilizzo di una memoria permanente che possa mantenere il valore memorizzato anche nel caso in cui il microcontrollore non dovesse essere più alimentato. Come detto è possibile fare questo per mezzo della memoria Programma. In particolare se il codice segreto deve essere fisso ed è noto sin dall'inizio, è sicuramente pratico far uso della memoria programma. Qualora però i dati devono essere variati durante la vita del programma, cambiare poche celle

---

<sup>22</sup> Microprocessori più complessi possiedono normalmente più accumulatori. Questo permette di ottimizzare i tempi di calcolo poiché non bisogna sempre caricare i dati in un solo accumulatore. La presenza di più accumulatori è inoltre fondamentale in architetture con pipeline.

<sup>23</sup> Se per esempio si sommano due numeri contenuti in un byte ed il risultato è salvato in un byte, il valore dovrà essere al massimo pari a 255. Se la somma dovesse generare un valore maggiore si avrà un overflow del registro.



---

di memoria singolarmente rende l'utilizzo della memoria flash, ovvero la memoria programma, non pratica. Per queste applicazioni i PIC di qualunque famiglia, tra cui i PIC18 mettono a disposizione la memoria EEPROM (Electrically Erasable Programmable Read Only Memory) in realtà dal momento che la cella di memoria può essere scritta singolarmente, la parola Read all'interno dell'acronimo EEPROM deve essere considerata come Read e Write. Diversamente dalla memoria flash, precedentemente descritta, non è necessario cancellare manualmente la cella di memoria, la cancellazione dei suoi contenuti, pur sempre necessaria, viene fatta in automatico all'avvio di un ciclo di scrittura.

La quantità di memoria EEPROM è ridotta a poche locazioni di memoria se paragonate alla memoria programma, sono infatti presenti solo 256 celle, ciononostante la sua presenza è certamente un bene prezioso<sup>24</sup>. Nel caso in cui dovesse essere necessaria più memoria EEPROM, è sempre possibile aggiungere molti KB per mezzo delle memorie EEPROM a soli 8 pin, controllabili via I2C o SPI<sup>25</sup>; la Microchip fornisce oltre ai PIC una delle più ampie scelte di memorie esterne I2C e SPI.

La memoria EEPROM, interna ai PIC, non è direttamente mappata né tra i Register File né nella Memoria Programma. La sua gestione avviene per mezzo di alcuni registri speciali dedicati (SFR):

- EECON1
- EECON2
- EEADR
- EEDATA

EECON1 è il registro utilizzato per il controllo della fase di scrittura e lettura. Tale registro è lo stesso che è necessario utilizzare per programmare la memoria flash, i bit CFGS e EEPGD determinano se il suo accesso è associato alla memoria Flash o EEPROM. Di seguito si riporta il contenuto del registro EECON1, tale rappresentazione è la stessa utilizzata nei datasheet della Microchip. Il registro EECON2 viene utilizzato per rendere la fase di scrittura in memoria sicura, ovvero evitare scritture accidentali della memoria. Infatti prima di avviare una sequenza di scrittura, è necessario scrivere nel registro EECON2 il valore 55h e AAh, in modo da dichiarare apertamente la volontà di avviare un processo di scrittura.

I registri EEADR e EEDATA sono rispettivamente utilizzati per scrivere l'indirizzo della locazione di memoria da andare a leggere o scrivere e il valore da assegnare o di ritorno dall'indirizzo selezionato<sup>26</sup>.

Vediamo ora in maggior dettaglio la sequenza d'istruzioni necessarie per scrivere e leggere la memoria EEPROM. Per ragioni di praticità e per introdurre qualche altra informazione utile alla programmazione, si mostrano le sequenze in assembler. In questo caso sono molto semplici e di facile comprensione anche se non si ha una conoscenza dell'assembler<sup>27</sup>. Nei Capitoli successivi si farà uso di una libreria scritta in C che ho personalmente scritto, ma che non fa altro che replicare la sequenza proposta in assembler.

---

<sup>24</sup> I PIC18, oltre alla protezione in lettura della memoria Programma, per permettere la protezione dell'IP (Intellectual Property) associato al programma, permette di proteggere anche la EEPROM da letture esterne effettuate per mezzo del programmatore.

<sup>25</sup> Un esempio della gestione di una EEPROM per mezzo del bus I2C verrà descritto nei successivi Capitoli.

<sup>26</sup> Il registro EEADR è un registro di un byte, da ciò discende anche il limite di 256 locazioni di memoria EEPROM. Naturalmente se si fosse utilizzato un registro più ampio o due registri si sarebbe potuta avere più memoria, ma i costi della memoria sono in generale piuttosto grandi, dunque se si vuole più memoria bisogna pagarla.

<sup>27</sup> Si rimanda al datasheet del PIC utilizzato per maggiori informazioni riguardo all'utilizzo delle istruzioni assembler.

## Registro EECON1: Data EEPROM Control Register 1

R/W-x	R/W-x	U-0	R/W-0	R/W-x	R/W-1	R/S-0	R/S-0
EEPGD	CFGFS	-	FREE	WRERR	WREN	WR	RD
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

### Leggenda

R = Readable bit  
-n = Value at POR

W = Writable bit  
1 = Bit is set

U = Unimplemented bit read as 0  
0 = Bit is cleared

S : Settable bit  
x = Bit is unknown

- Bit 7**      **EEPGD** : Selezione memoria Programma (flash) o EEPROM  
1 : Seleziona accesso memoria Programma  
0 : Seleziona accesso memoria EEPROM
- Bit 6**      **CFGFS** : Memoria Programma/EEPROM o configurazione  
1 : Accedi ai registri di configurazione  
0 : Accedi alla memoria Programma o EEPROM
- Bit 5**      Non implementato, letto come 0.
- Bit 4**      **FREE** : Flash Row Erase Enable bit  
1 : Cancella la sezione di memoria Programma al prossimo ciclo di scrittura  
0 : Effettua solo la scrittura
- Bit 3**      **WRERR** : Bit di errore scrittura memoria Programma o EEPROM  
1 : Il ciclo di scrittura è stato prematuramente terminato  
0 : Il ciclo di scrittura è stato propriamente terminato
- Bit 2**      **WREN** : Abilita la possibilità di scrittura  
1 : Abilita la scrittura nella memoria Programma o EEPROM  
0 : Disabilita la scrittura nella memoria Programma o EEPROM
- Bit 1**      **WR** : Bit di controllo della scrittura  
1 : Avvia la scrittura. Il bit viene posto a 0 a fine scrittura.  
0 : Il ciclo di scrittura è terminato
- Bit 0**      **RD** : Bit di controllo della lettura  
1 : Avvia il ciclo di lettura  
0 : Non avvia il ciclo di lettura

## Sequenza di lettura

Dalla sequenza di lettura è possibile vedere che è possibile scrivere i commenti al fianco dell'istruzione, precedendo quest'ultimi con un punto e virgola. Il primo passo consiste nel caricare il valore dell'indirizzo di cui si vuole leggere il contenuto. In C si sarebbe scritto solamente `EEADR = indirizzo` mentre in assembler tutti i movimenti tra registri vengono generalmente fatti per mezzo del registro intermedio W, ovvero l'accumulatore. Tale passaggio sarebbe anche il modo con cui il compilatore traduce la nostra istruzione in C.

```
MOLW    DATA_EE_ADDR    ; Carico l'indirizzo da leggere in W
MOVWF   EEADR             ; Carico l'indirizzo da leggere in EEADR
BCF     EECON1, EEPGD     ; Pongo a 0 il bit EEPGD di EECON1
BCF     EECON1, CFGFS     ; Pongo a 0 il bit CFGFS di EECON1
BSF     EECON1, RD        ; Pongo a 1 RD, avviando la lettura
MOVF    EEDATA, W         ; Leggo il contenuto di EEDATA
```

Una volta caricato il valore del nostro indirizzo in EEADR, è necessario impostare i bit EEPGD e CFGFS. Quando il loro valore è 0 vuol dire che la fase di read è associata alla EEPROM e non alla memoria flash. Per porre a 0 tali bit ci sono vari modi, ma sicuramente il più pratico è quello di usare l'istruzione BCF (Bit Clear f) che pone appunto a 0 (clear) il bit di un determinato registro. Impostati i bit di selezione per la EEPROM è possibile avviare la lettura per mezzo del bit RD che dovrà essere posto ad 1. Per fare questo si utilizza l'istruzione complementare a BCF ovvero BSF. Avviata la lettura, il dato verrà reso disponibile all'interno del registro EEDATA a partire dal ciclo istruzione successiva, ovvero è

possibile leggere EEDATA subito dopo. Il registro EEDATA mantiene il dato fino ad una nuova lettura o eventuale scrittura su di esso. Come visibile, anche questa volta il valore viene letto per mezzo del registro W. Una volta posto il dato in W sarà possibile porlo in un qualunque altro registro presente nella memoria dati, ovvero in una qualunque nostra variabile.

## Sequenza di scrittura

La sequenza di scrittura diversamente da quella di lettura è un po' più laboriosa, per tale ragione ho deciso di scrivere una libreria ad hoc per semplificare la fase lettura e di scrittura in EEPROM.

```

MOLW    DATA_EE_ADDR    ; Carico l'indirizzo da scrivere in W
MOVWF   EEADR            ; Carico l'indirizzo da scrivere in EEADR
MOLW    DATA_EE_DATA    ; Carico il dato da scrivere in W
MOVWF   EEDATA           ; Carico l'indirizzo da leggere in EEADR
BCF     EECON1, EEPGD     ; Pongo a 0 il bit EEPGD di EECON1
BCF     EECON1, CFGS      ; Pongo a 0 il bit CFGS di EECON1
BSF     EECON1, WREN      ; Abilito la fase di scrittura (non avvio)

BCF     INTCON, GIE       ; Disabilito gli Interrupt

MOLW    55h              ; Carico 55h in W
MOVWF   EECON2           ; Carico il dato in EECON2
MOLW    AAh              ; Carico AAh in W
MOVWF   EECON2           ; Carico il dato in EECON2

BSF     EECON1, WR        ; Avvio la fase di scrittura
BSF     INTCON, GIE       ; Abilito nuovamente gli Interrupt
BCF     EECON1, WREN      ; Disabilito la scrittura

```

La parte iniziale della sequenza è praticamente identica alla precedente, però oltre che ad impostare il registro dell'indirizzo EEADR è necessario impostare anche il registro EEDATA con il dato che si vuole scrivere. Fatto questo si seleziona la EEPROM piuttosto che la memoria flash, per mezzo dei bit EEPGD e CFGS. In ultimo si abilita la possibilità di scrivere la memoria EEPROM settando il valore del bit WREN<sup>28</sup>.

Una volta abilitata la possibilità di scrivere in EEPROM è consigliato disabilitare le interruzioni (se usate), se infatti la sequenza di scrittura dovesse essere interrotta si potrebbe avere perdita di dati o dati corrotti. Disabilitare le interruzioni è necessario scrivere la sequenza magica all'interno del registro EECON2 ovvero prima il byte 55h e successivamente AAh. Come visibile tale scrittura avviene sempre per mezzo del registro W. Dopo tale sequenza è possibile avviare la scrittura ponendo ad 1 il bit WR nel registro EECON1. Tale bit rimarrà ad 1 per tutto il processo di scrittura, che diversamente dalla lettura può impiegare più di un ciclo di clock. In particolare se si ha interesse a sapere la fine del processo di scrittura si può controllare in polling (ovvero leggendo ripetutamente) il bit WR. Se ci sono errori in scrittura, ovvero la scrittura non è stata portata a termine, viene settato il bit WRERR. Tale errore non dice se il contenuto è quello da noi voluto, ma solo se la fase di scrittura è terminata prematuramente. Per controllare che la fase di scrittura è avvenuta propriamente è bene controllare il dato per mezzo di una sequenza di lettura dell'indirizzo stesso. Tale

<sup>28</sup> Oltre alla sequenza 55h e AAh da scrivere nel registro EECON2, un'altra tecnica per proteggere la EEPROM da scritture accidentali è data dal bit WREN. In particolare all'avvio del PIC (Power Cycle) tale bit viene posto a 0. Inoltre la scrittura in EEPROM durante la fase di Power-Up è disabilitata. In questo modo si evita di compromettere i dati in EEPROM a causa di scritture accidentali.

---

approccio è consigliato soprattutto in quelle applicazioni in cui ogni cella di memoria verrà utilizzata a tal punto che il numero massimo di scritture può essere raggiunto. In ogni modo un controllo del dato scritto, rimane una buona pratica di programmazione a prescindere da tale problema.

## L'oscillatore

Ogni microcontrollore richiede, per il suo funzionamento, di un clock, ovvero di quel segnale digitale per mezzo del quale vengono scandite tutte le operazioni. Come illustrato già in precedenza c'è un legame diretto tra le istruzioni eseguite ed il clock. Il segnale di clock viene generato per mezzo di un oscillatore. I PIC18 possiedono al loro interno tutta la circuiteria per generare un clock interno ma anche la circuiteria per realizzare un clock per mezzo di un quarzo esterno. La complessità della circuiteria associata all'oscillatore è più o meno complessa a seconda del PIC utilizzato.

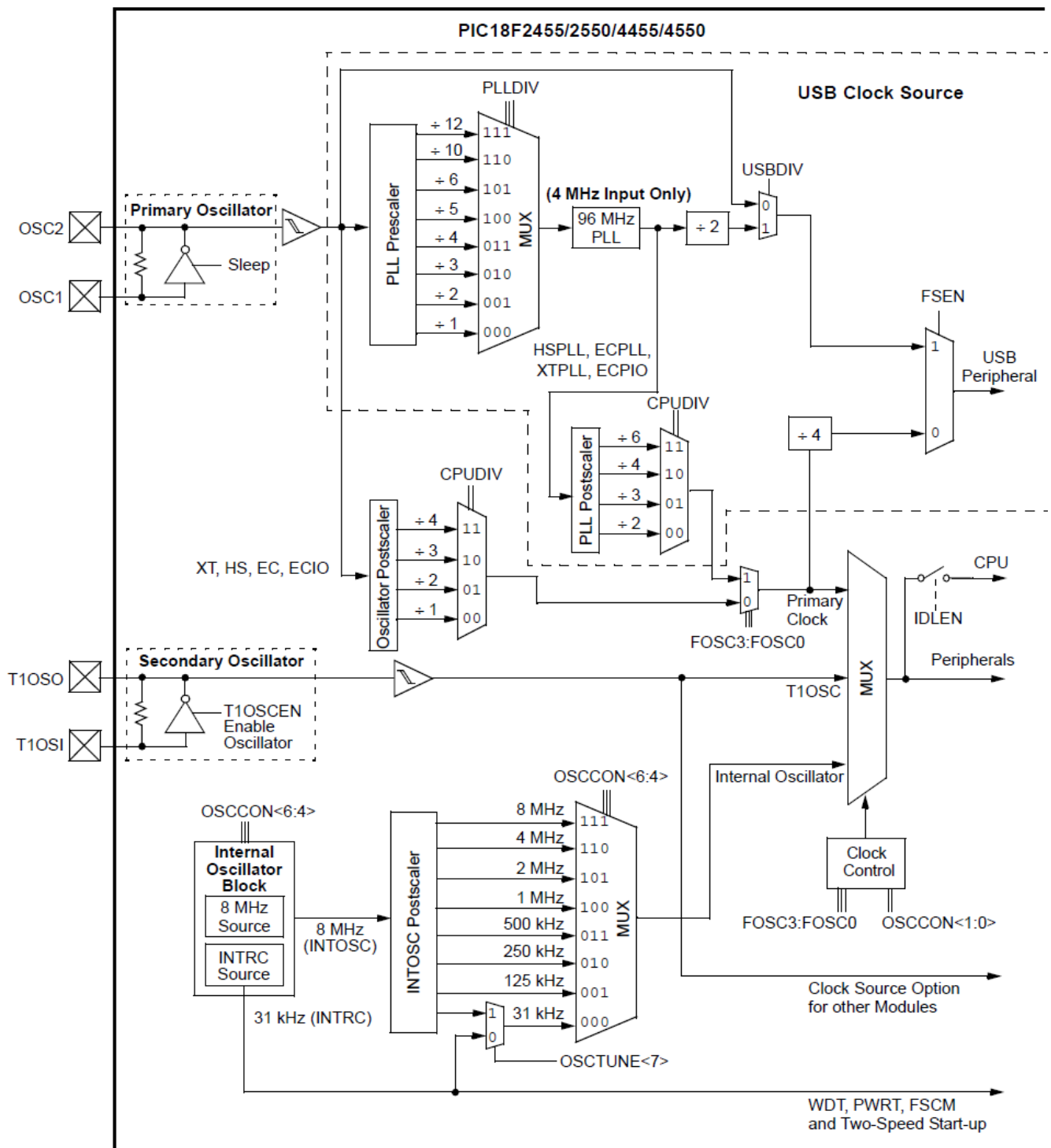


Figura 7: Ciruiteria di clock del PIC18F4550

---

Nel nostro caso, avendo preso come riferimento il PIC18F4550 è bene mettere subito in evidenza che avremo a che fare con una circuiteria un po' complessa. Questo discende dal fatto che il PIC18F4550 possiede al suo interno il modulo USB. Al fine di rendere il PIC compatibile sia con le specifiche USB per dispositivi a bassa ed alta velocità, le specifiche del clock sono state tanto stringenti da richiedere alcuni moduli aggiuntivi (un postscaler e un prescaler). In Figura 7 è riportato lo schema a blocchi della circuiteria associata all'oscillatore del PIC18F4550. E' possibile subito vedere che sono presenti diverse selezioni e percorsi di cui si parlerà a breve.

Prima di entrare nel dettaglio dello schema a blocchi è bene mettere in evidenza il fatto che il PIC18F4550 può avere il proprio oscillatore (o meglio oscillatori) impostabile in 12 diversi modalità. Dallo schema a blocchi di Figura 7 è possibile vedere che sono presenti due oscillatori nominati primario e secondario. Il primario è normalmente utilizzato per permettere l'oscillazione di un quarzo esterno per generare il clock utilizzato dal microcontrollore. Questo è valido in generale per i PIC18, per il PIC18F4550 l'oscillatore primario viene anche utilizzato per generare il clock per il modulo USB, quando questo viene abilitato. Oltre a questi due oscillatori, è presente l'oscillatore interno ad 8MHz. Quando abilitato può essere utilizzato per generare il clock del microcontrollore. E' possibile notare che la sua uscita è collegata ad un postscaler (divisore di frequenza), per mezzo del quale è possibile ottenere una serie di frequenze interne che è possibile selezionare per mezzo del MUX interno. Sulla destra dello schema a blocchi è possibile osservare altri due MUX, per mezzo dei quali è possibile selezionare il clock da utilizzare per le periferiche e se utilizzare l'oscillatore principale per il modulo USB. Un'altra caratteristica importante presente in molti PIC18 è la presenza del PLL (Phase Lock Loop) interno. Tale dispositivo permette di ottenere clock ad elevata frequenza, partendo da quarzi a frequenza ridotta. Il diagramma a blocchi, come detto, mostra la presenza di diversi oscillatori. Questi possono essere impostati per operare in diverse modalità, alcune di queste vengono selezionate per mezzo dei registri CONFIG1L e CONFIG1H. Tali impostazioni vengono settate nel PIC durante la fase di programmazione e rimangono settate allo stesso modo fino a quando il PIC non viene riprogrammato. Altre impostazioni sono invece impostabili "on the fly", ovvero durante l'esecuzione del programma stesso, il registro da modificare è OSCCON. Questo registro permette di selezionare la sorgente del clock da utilizzare come base tempo<sup>29</sup>. I modi operativi dell'oscillatore del PIC18F4550 sono, come detto 12. Questi possono essere selezionati per mezzo dei registri CONFIG.

- **XT**

La modalità XT utilizza l'oscillatore primario al quale deve essere collegato un quarzo.

- **XTPLL**

E' come la modalità XT ma con il PLL abilitato.

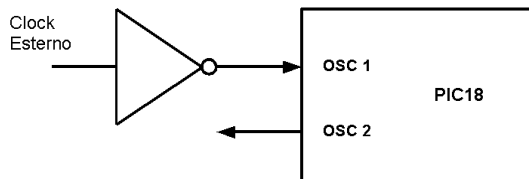
- **HS**

E' come la modalità XT ma viene utilizzata nel caso di quarzi ad alta frequenza. Tale modalità può però essere utilizzata anche senza quarzo, facendo uso di un oscillatore esterno, come riportato in Figura 8. Quando si fa utilizzo di un oscillatore esterno, non vi è il tipico problema di ritardo legato allo start up (avvio) dell'oscillatore. Tale ritardo può anche essere associato ad un risveglio del microcontrollore, oltre che ad un power up del PIC o del sistema. Il pin di uscita risulta molto utile nel caso si vogliano sincronizzare altri dispositivi o effettuare delle calibrazioni, senza influenzare

---

<sup>29</sup> Vi sono anche altri registri associati alle impostazioni dell'oscillatore. Per maggiori dettagli si rimanda al datasheet del PIC utilizzato.

l'oscillatore principale.



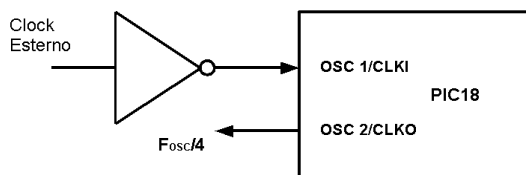
**Figura 8:** Modalità HS con oscillatore esterno

- **HSPLL**

E' come la modalità HS ma viene abilitato il PLL interno.

- **EC**

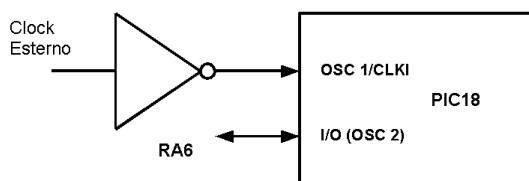
In questa modalità un clock esterno viene fornito in ingresso al PIC, ovvero non è richiesto il quarzo sull'oscillatore principale. Il secondo piedino normalmente utilizzato per il quarzo viene utilizzato per riportare la  $F_{osc}/4$  in uscita. Lo schema a blocchi è riportato in Figura 9.



**Figura 9:** Modalità EC con oscillatore esterno

- **ECIO**

Come la modalità EC, ma il pin RA6 può essere utilizzato come standard I/O invece che  $F_{osc}/4$ . Lo schema a blocchi è riportato in Figura 10.



**Figura 10:** Modalità EC con oscillatore esterno

- **ECPLL**

Come la modalità EC ma con il PLL interno abilitato.

- **ECPIO**

Come la modalità ECIO ma con il PLL interno abilitato.

- **INTHS**

L'oscillatore interno è utilizzato come clock per il microcontrollore, mentre l'oscillatore HS è utilizzato come sorgente per il clock utilizzato per il modulo USB.

- **INTXT**

L'oscillatore interno è utilizzato come clock per il microcontrollore, mentre l'oscillatore XT è utilizzato come sorgente per il clock utilizzato per il modulo USB.

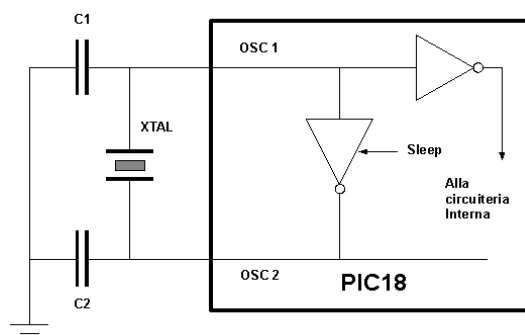
- **INTIO**

L'oscillatore interno è utilizzato come clock per il microcontrollore, mentre l'oscillatore EC è utilizzato come sorgente per il clock utilizzato per il modulo USB. Il pin RA6 è utilizzato come I/O standard.

- **INCKO**

L'oscillatore interno è utilizzato come clock per il microcontrollore, mentre l'oscillatore EC è utilizzato come sorgente per il clock utilizzato per il modulo USB.

Vediamo in maggior dettaglio le modalità HS, XT, HSPLL e XTPLL. In tutte queste modalità è necessario un quarzo, la configurazione utilizzata è riportata in Figura 11. E' possibile notare che oltre al quarzo sono presenti anche due condensatori.



**Figura 11:** *Utilizzo del quarzo esterno*

La circuiteria potrebbe essere anche più complessa a seconda delle esigenze e del quarzo utilizzato. I condensatori C1 e C2 sono dello stesso valore e vanno scelti a seconda del quarzo utilizzato. In progetti che devono essere rilasciati per applicazioni professionali, è sempre bene fare test sulla qualità dell'oscillazione, al variare della temperatura e tensione di alimentazione, nel range dei valori per cui deve essere specificato il sistema. La Microchip fornisce delle linee guida che permettono di selezionare i condensatori per alcuni valori standard di cristalli, in Tabella 2 sono riportati i valori presenti sul datasheet.

Modalità	Frequenza	C1 e C2
XT	4 MHz	27 pF
HS	4 MHz	27 pF
HS	8 MHz	22 pF
HS	20 MHz	15 pF

**Tabella 2:** *Valori tipici di C1 e C2*

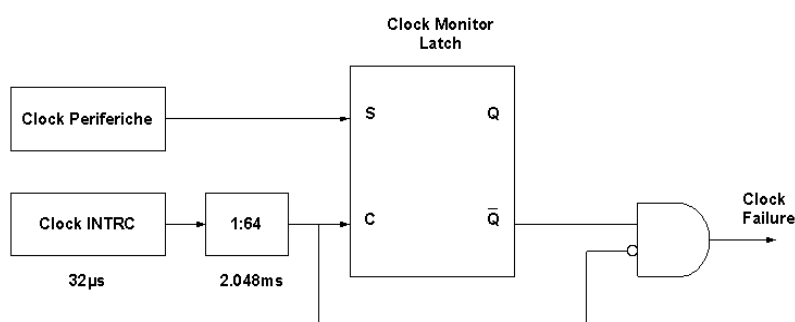


È bene mettere in evidenza che tali modalità sono seguite da un postscaler (divisore) che può essere utilizzato per impostare valori di frequenza di clock inferiori a quello del quarzo stesso. Il valore può essere impostato per mezzo del registro interno CPUDIV.

Da quanto ora descritto, si è capito che sono presenti diverse possibilità per generare il clock utilizzato dal microcontrollore. Nel caso specifico del PIC18F4550 ci sono anche altre opzioni di clock specificatamente pensate per il modulo USB. In particolare è bene mettere in evidenza che il modulo USB a seconda delle modalità supportate (low o high speed) richiede rispettivamente una frequenza operativa di 6MHz e 48MHz. Nel caso della modalità low speed è obbligatorio far uso di un quarzo esterno da 24MHz mentre nel caso high speed si ha maggior flessibilità. Poiché è richiesto di avere una frequenza di 48MHz per il modulo USB è necessario far uso del modulo PLL interno precedentemente descritto.

Qualora il clock sia generato dalla stessa sorgente, il modulo USB può operare a 48MHz, mentre per mezzo dei postscaler è possibile impostare la frequenza del clock del microcontrollore a valori inferiori. Una caratteristica introdotta dalla Microchip nei PIC18 è quella di poter cambiare anche la frequenza di clock e sorgente (oscillatore primario, secondario o interno) durante l'esecuzione del programma, questo può essere fatto per mezzo dei bit SCS1:SCS0 del registro OSCCON. Tale caratteristica è stata fornita prevalentemente per ragioni associate all'efficienza del dispositivo, infatti abbassando la frequenza di clock è possibile ridurre la potenza dissipata dal microcontrollore. Questo significa che in condizioni in cui non è richiesta alta velocità, è possibile ridurre i consumi riducendo la frequenza o la sorgente di clock.

Un'altra caratteristica importante dei PIC18 è la presenza di un clock secondario in caso di malfunzionamento del clock delle periferiche (Fail Safe Clock Monitor). Qualora il clock associato alle periferiche si dovesse interrompere sia esso il primario, secondario o interno, il clock viene cambiato in automatico sul clock di guardia ottenuto dall'oscillatore INTRC. Normalmente questo clock risulta di frequenza più bassa di quella normalmente utilizzata, dunque particolari applicazioni potrebbero non funzionare più (il modulo USB è tra queste), ciononostante tale opzione risulta particolarmente importante poiché permette di abilitare funzioni di emergenza o effettuare uno spegnimento del sistema controllato. Lo schema a blocchi di tale funzione è riportato in Figura 12.



**Figura 12:** Schema a blocchi del monitoraggio del clock

Per abilitare tale funzione è necessario settare il bit FCMEN del registro CONFIG1H.

## Power Managment

In un mondo che richiede sempre più energia, avere la possibilità di ridurre i consumi è sicuramente un'opzione ben gradita. Le nuove tecnologie grazie alla miniaturizzazione dei processi di realizzazione dei circuiti integrati, nonché della diminuzione delle tensioni operative, ha permesso di ridurre notevolmente i consumi. Oltre alle innovazioni tecnologiche, accorgimenti più semplici possono essere altrettanto validi quale opzione per ridurre i consumi. In particolare una tecnica utilizzata è quella di utilizzare una frequenza di clock non più alta del necessario, infatti all'aumentare della frequenza di clock aumentano anche la potenza richiesta dal sistema. I PIC18 mettono a disposizione, come si vedrà a breve nei prossimi paragrafi, la circuiteria necessaria per cambiare la frequenza di clock anche durante l'esecuzione del programma. Oltre alla possibilità di variare la frequenza di clock vi sono altre opzioni e tecniche utilizzate per la riduzione dei consumi. Il PIC18F4550 possiede sette modalità operative differenti, per mezzo delle quali è possibile gestire in maniera ottimale i consumi, in accordo con le specifiche del progetto. Le varie modalità si raggruppano in tre diversi stati in cui il microcontrollore può trovarsi:

- Modalità Run
- Modalità Idle
- Modalità Sleep

Nella modalità Run sia la CPU che le periferiche sono attive, il variare della frequenza di clock permette di controllare i consumi. Nella modalità Idle, la CPU è disattivata mentre le periferiche possono rimanere abilitate a seconda delle esigenze. In questa modalità è possibile ottenere consumi ridotti fino a  $5.8\mu A$ . Nella modalità Sleep, sia la CPU che le periferiche vengono disabilitate riducendo ulteriormente i consumi a  $0.1\mu A$ . Dal momento che sia nella modalità Idle che Sleep la CPU viene disattivata, si capisce che non viene eseguita nessuna istruzione. Da ciò discende il fatto che l'unico modo per riattivare la CPU è per mezzo di un Reset o di un interrupt dalle periferiche (preventivamente abilitato), in particolare il Reset può anche essere generato dal Watchdog Timer (descritto a breve). Le sette differenti modalità riportate in Tabella 3 è possibile abilitarle per mezzo del bit IDLEN presente nel registro OSCCON e i bit SCS1:SCS0 presenti ancora nel registro OSCCON.

Mode	IDLEN	SCS1:SCS0	Stato CPU	Stato Periferiche	Clock disponibile e sorgente
PRI_RUN	N/A	00	ON	ON	Modalità primaria, tutte le sorgenti clock sono disponibili.
SEC_RUN	N/A	01	ON	ON	Modalità secondaria, il clock dal Timer1 è disponibile.
RC_RUN	N/A	1x	ON	ON	Oscillatore RC interno genera il clock.
PRI_IDL	1	00	OFF	ON	Modalità primaria, tutte le sorgenti clock sono disponibili.
SEC_IDL	1	01	OFF	ON	Modalità secondaria, il clock dal Timer1 è disponibile.
RC_IDL	1	1x	OFF	ON	Oscillatore RC interno genera il clock.
Sleep	0	N/A	OFF	OFF	Nessuno, tutte le sorgenti sono disabilitate

**Tabella 3:** Modalità risparmio energia disponibili

Per poter entrare nella modalità Idle o Sleep, è necessario eseguire l'istruzione SLEEP, poi, a seconda del valore del bit IDLEN, il microcontrollore entrerà nello stato di Sleep o Idle; i bit

---

SCS1:SCS0 determinano poi la sorgente di clock. Si fa subito notare che in caso si voglia utilizzare la modalità Idle o Sleep, dal momento che dopo l'esecuzione dell'istruzione SLEEP non verranno più eseguite ulteriori istruzioni, i bit di competenza della modalità d'interesse dovranno essere preventivamente impostati come richiesto. Vediamo qualche dettaglio in più sulle diverse modalità:

- **Modalità PRI\_RUN**

Questa modalità rappresenta quella principale che viene impostata dopo il Reset. Il clock disponibile è quello selezionato per mezzo dei bit FOSC3:FOSC0 del registro di configurazione CONFIG1H.

- **Modalità SEC\_RUN**

Questa modalità rappresenta quella compatibile con gli altri microcontrollori PIC18 che possiedono la modalità di risparmio energia. Questa viene imposta cambiando il valore dei bit SCS1:SCS0 al valore 00. Il clock disponibile è il Timer1.

- **Modalità RC\_RUN**

Questa modalità rappresenta quella a maggior risparmio energetico, dal momento che fa utilizzo della frequenza di clock più bassa, generata per mezzo dell'oscillatore RC interno. Sia la CPU che le periferiche prelevano il clock dal INTOSC multiplexer, inoltre il clock primario viene disattivato, riducendo ulteriormente i consumi<sup>30</sup>. Tale modalità viene selezionata impostando i bit SCS1:SCS0 ad 10<sup>31</sup>.

- **Modalità PRI\_IDL**

Questa modalità è la prima dello stato Idle ed è quella che permette un minor risparmio energetico tra le modalità Idle. Questo discende dal fatto che, pur disattivando la CPU, le periferiche continuano ad essere alimentate e provviste del clock principale. Questo se da un lato mantiene i consumi associati al clock primario, permette di passare dallo stato Idle allo stato RUN in maniera veloce, visto che l'oscillatore primario non deve essere riattivato. Per entrare in questa modalità bisogna settare il bit IDLEN, impostare i bit SCS1:SCS0 al valore 00 ed eseguire l'istruzione di SLEEP. Al risveglio del microcontrollore la CPU utilizza come clock l'oscillatore primario.

- **Modalità SEC\_IDL**

Questa modalità, diversamente dalla precedente, disattiva il clock primario, oltre alla CPU. Il clock fornito alle periferiche è ricavato dal Timer1. Per entrare in questa modalità bisogna settare il bit IDLEN, impostare i bit SCS1:SCS0 al valore 01 ed eseguire l'istruzione di SLEEP. Al risveglio del microcontrollore la CPU utilizza come clock il Timer1.

- **Modalità RC\_IDL**

Questa modalità, è quella a maggior risparmio energetico tra le modalità Idle, poiché è possibile impostare il clock a più bassa frequenza. La CPU, come per ogni modalità Idle, viene disattivata. Il clock fornito alle periferiche è quello generato dall'INTOSC multiplexer. Per entrare in questa modalità bisogna settare il bit IDLEN, impostare i bit SCS1:SCS0 al valore 01 ed eseguire l'istruzione di SLEEP. Al risveglio del

---

<sup>30</sup> Dal momento che il clock primario viene disattivato, quando si avesse la necessità di passare nuovamente al clock primario, si avrà del ritardo dovuto all'attivazione e la stabilizzazione dello stesso. Tale ritardo è lo stesso che si verrebbe ad avere con un power cycle.

<sup>31</sup> In realtà per impostare tale modalità sia il valore 10 che 11 verrebbero accettati. Però, al fine di mantenere eventuali compatibilità con versioni successive di PIC si raccomanda di utilizzare il valore 10 piuttosto che 11.

---

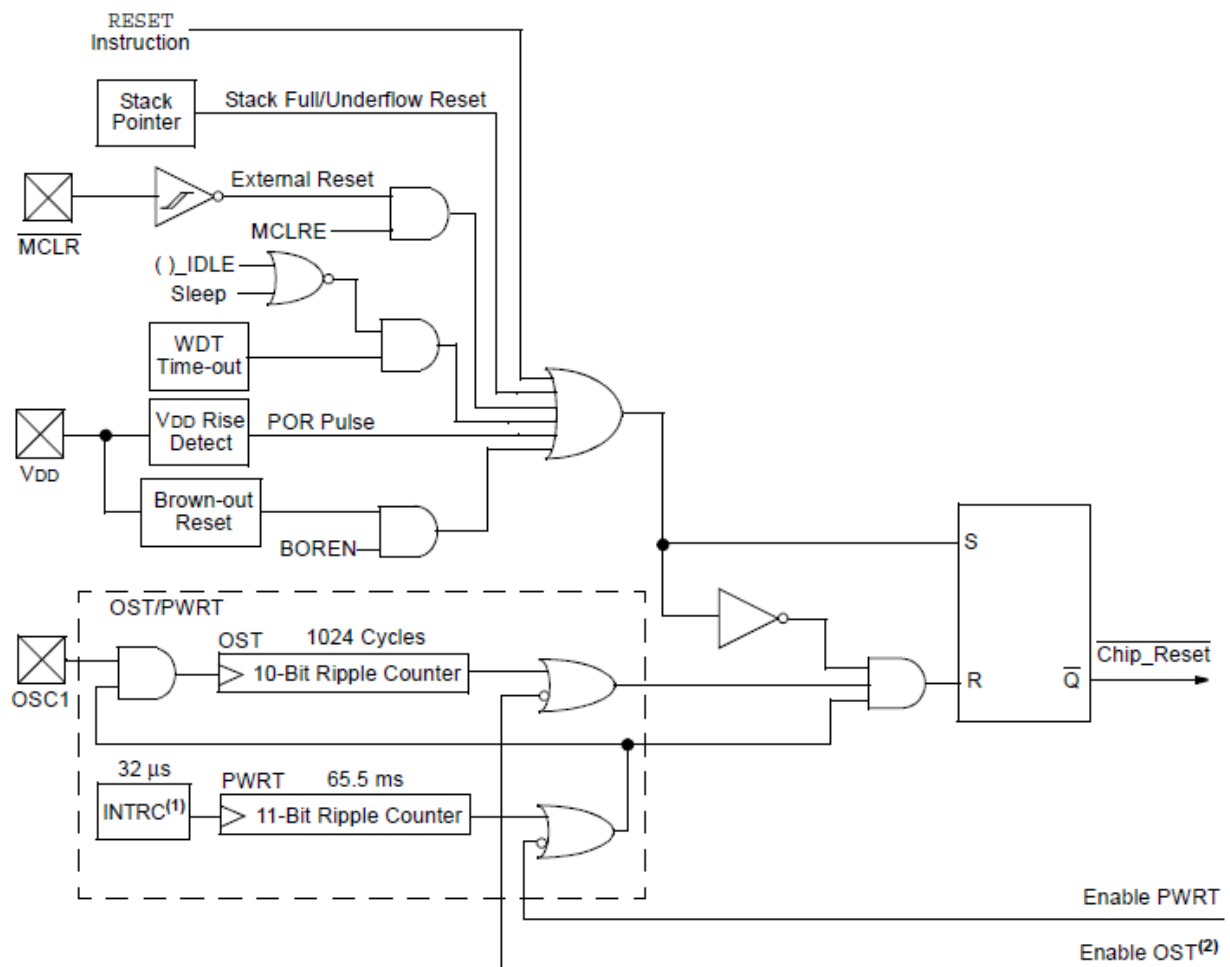
microcontrollore la CPU utilizza come clock l'oscillatore INTOSC.

- **Modalità Sleep**

Questa modalità rappresenta la modalità a maggior risparmio energetico poiché tutte le periferiche e la CPU vengono disattivate. Se il Watchdog Timer era precedentemente attivato il suo clock verrà mantenuto attivo garantendo in questo modo l'evento di time out del Watchdog. Tale evento, assieme all'evento di Reset e interrupt da periferiche, permette il risveglio del microcontrollore. Il clock che verrà utilizzato al risveglio dipenderà dal valore dei bit SCS1:SCS0 prima dell'attivazione della modalità Sleep.

## Circuiteria di Reset

Come si è visto nella parte relativa all'organizzazione della memoria, il programma verrà eseguito a partire dall'indirizzo del Reset Vector. L'evento di Reset può essere considerato un tipo d'interruzione per la CPU ad alta priorità, in particolare la CPU non può ignorare una richiesta di Reset. Ogni microcontrollore, come anche microprocessore possiede una circuiteria interna di Reset. Nonostante l'operazione di Reset sia apparentemente semplice, ovvero portare il Program Counter nella posizione del Reset Vector, la circuiteria associata al Reset è relativamente più complicata di quanto non si possa pensare. La circuiteria di Reset del PIC18F4550 è riportata in Figura 13.



**Figura 13:** Circuiteria di Reset del PIC18F4550

La ragione della sua complessità è legata al fatto che i PIC18 possiedono differenti sorgenti che possono generare lo stato di Reset della CPU. Lo stato di Reset oltre che a cambiare il valore del Program Counter determina anche il cambio del valore di alcuni registri interni. In particolare il datasheet riporta sempre le tabelle con i vari registri e il valore assunto dai vari bit in caso di un Reset della CPU. Alcune volte i bit vengono posti a 0 o 1, mentre altre volte assumono un valore casuale. Ritornando alla Figura 13 è possibile osservare che il segnale di Reset rappresenta il risultato di operazioni logiche più o meno complesse, ma fondamentalmente è presente una OR con le varie possibilità di Reset e una AND che disabilita le funzioni di Reset nel caso in cui non sia trascorso un tempo minimo dall'attivazione del microcontrollore. I vari eventi che possono causare la generazione di un

---

Reset sono<sup>32</sup>:

- Power On Reset (POR)
- Master Clear Reset (MCLR)
- Programmable Brown-out Reset (BOR)
- Watchdog Timer (WDT)
- Stack pieno
- Stack vuoto
- Istruzione RESET

Per sapere quale tipo di evento ha generato il Reset, è bene andare a leggere il valore del registro RCON. Vediamo ora in che modo la CPU può essere resettata.

## Power On Reset

Il Power On Reset (POR) rappresenta uno dei primi eventi che si viene a verificare una volta che si alimenta il sistema. Una volta passato un tempo minimo dal momento in cui viene rilevata la tensione minima operativa, viene attivato un Reset automatico, in maniera da portare i registri interni al PIC in uno stato noto. In realtà stato noto non significa necessariamente sapere il valore di tutti i registri; per sapere il valore iniziale dei registri è bene sempre far affidamento al datasheet del PIC utilizzato.

La ragione per cui è necessario attendere un tempo minimo è legata al fatto che prima di iniziare una qualunque operazione, è necessario che l'oscillatore si sia stabilizzato. In questo modo è possibile garantire che le varie istruzioni verranno eseguite correttamente e soprattutto nello stesso tempo base. Per permettere il corretto funzionamento della circuiteria interna POR è necessario collegare un resistore compreso tra 1Kohm e 10Kohm tra il pin MCLR e Vcc.

## Master Clear Reset

Il Master Clear Reset (MCLR) è la tecnica principale per mezzo della quale è possibile resettare manualmente il PIC. Il segnale MCLR è riportato in uscita al pin nominato appunto MCLR. Tale pin, se collegato a massa resetta il PIC; per tale ragione viene normalmente collegato ad un pulsante verso massa. Dal momento che tale pin viene anche utilizzato per la programmazione on board, al fine di non danneggiare il programmatore, in caso di pressione accidentale del tasto di Reset durante la fase di programmazione, il tasto di Reset è frequentemente collegato a massa per mezzo di un resistore.

La circuiteria associata al pin MCLR possiede al suo interno un filtro anti-rimbalzo, evitando che spike accidentali possano resettare la CPU. In schemi elettrici è frequente vedere la linea MCLR collegata a VCC per mezzo di un resistore di 10Kohm, tale resistore viene in realtà utilizzato dalla circuiteria POR (Power On Reset). Il resistore può avere un valore compreso tra 1Kohm e 10Kohm.

Nel caso in cui si dovesse essere a corto di ingressi, il pin MCLR potrebbe anche essere impostato come input, perdendo però la funzione di resettare la CPU dall'esterno. Personalmente sconsiglio di utilizzare il pin MCLR per altre funzioni, a meno di forti ragioni pratiche in cui un tasto di Reset non servirebbe a molto di più di un power-cycle (ovvero accendi e spegni il sistema).

---

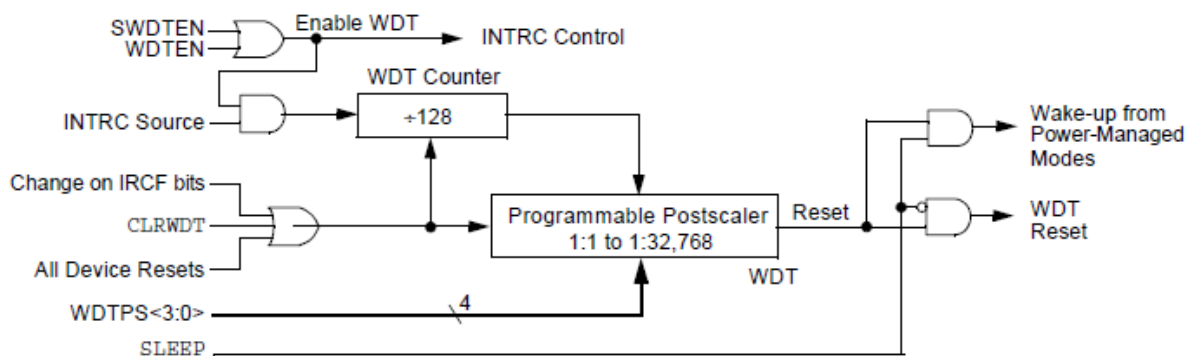
<sup>32</sup> Sono in realtà presenti anche altri tipi di Reset, ma questi sono quelli presenti su tutti i PIC18.

## Programmable Brown-out Reset (BOR)

Tale funzione è particolarmente utile poiché permette di resettare il PIC qualora la tensione vada al disotto di una soglia stabilita. La sua utilità la si apprezza specialmente in sistemi in cui l'affidabilità delle misure viene ad essere ridotta in caso in cui la tensione non è sufficiente. Naturalmente tali situazioni possono essere gestite anche in altro modo. La circuiteria BOR può funzionare in varie modalità selezionabili per mezzo dei bit BOREN1:BOREN0 del registro CONFIG2L. Normalmente la modalità Hardware è selezionata di Default. Oltre a tali bit di controllo sono presenti i bit BORV1:BORV0 sempre del registro CONFIG2L, per mezzo dei quali è possibile selezionare la soglia alla quale deve avvenire il Reset. La tensione di riferimento utilizzata per il confronto è quella generata dal generatore band gap da 1.2V.

## Watchdog Timer (WDT)

Il Watchdog, letteralmente cane da guardia, rappresenta un timer utilizzato per resettare la CPU qualora qualcosa dovesse andare male. Per il suo principio di funzionamento però potrebbe non rilevare tutti i casi di anomalia. Lo schema a blocchi della sua circuiteria è riportato in Figura 14.



**Figura 14:** Circuiteria di Reset del PIC18F4550

Il suo funzionamento è molto semplice, fondamentalmente il Watchdog è un timer che viene sempre incrementato, qualora dovesse arrivare a fine conteggio resetta la CPU. Si capisce che, se il Watchdog viene abilitato, al fine di evitare il Reset della CPU è necessario sempre azzerare il conteggio. L'idea dietro è legata al fatto che periodicamente devo azzerare il conteggio, dunque se il programma dovesse entrare in un loop vizioso o entrare in stallo, il Watchdog continuerebbe il suo conteggio fino a resettare la CPU. La base tempi del Watchdog è fissa ed è rappresentata dall'oscillatore INTRC. Tale oscillatore è indipendente dall'oscillatore principale, dunque continua il suo funzionamento anche se il microcontrollore è in stato di Sleep o Idle. Come visibile in Figura 14 il clock rappresentato da INTRC dopo essere andato al Watchdog, passa per il Postscaler (un divisore), ovvero un secondo contatore per mezzo del quale ritardare ulteriormente il conteggio principale del Watchdog. Sempre da Figura 14 è inoltre possibile vedere che il Watchdog e il Postscaler vengono azzerati nel caso in cui viene eseguita l'istruzione CLRWDT, un Reset generale, o un cambio dei bit IRCF (associati al cambio della frequenza operativa del microcontrollore).

---

## Stack Reset

Il Reset associato allo Stack può risultare particolarmente utile per ripristinare il corretto funzionamento del sistema. Si ricorda infatti che lo Stack è rappresentato da quell'insieme di registri il cui compito è di memorizzare l'indirizzo di ritorno da una chiamata ad una funzione, sia essa associata ad un interrupt o semplicemente una funzione standard. Si capisce che se lo Stack dovesse avere un evento di overflow, ovvero di traboccamento, vorrebbe dire che delle informazioni di ritorno verrebbero perse, portando prima o poi il sistema in uno stato non voluto. Lo stesso potrebbe accadere in caso di underflow. Il Reset associato allo Stack viene abilitato per mezzo del bit STVREN del registro CONFIG4L. Se abilitato (posto ad 1), un eventuale overflow o underflow dello Stack, genera un Reset del sistema.

Nel caso in cui il Reset dello Stack non dovesse essere abilitato, i bit STKFUL e STKUNF vengono settati di conseguenza. Tali bit sono presenti nel registro STKPTR e vengono resettati o via software o successivamente ad un POR Reset. In sistemi ad alta affidabilità lo Stack Reset permette di ripristinare il normale funzionamento del sistema, ciononostante eventuali Reset dovuti a problemi dello Stack, possono essere indice di problemi software. Questo risulta particolarmente vero in caso in cui si sia scritto il software in assembler.

## Istruzione RESET

Tutti i vari eventi finora descritti sono più o meno legati ad eventi hardware. L'ultimo caso qui descritto è in realtà di tipo software. Tra le varie istruzioni che il PIC18 può eseguire vi è quella di `RESET` ovvero l'istruzione che permette di resettare la CPU per mezzo del codice stesso. Questa istruzione, una volta eseguita ha gli stessi effetti di un Reset Hardware. Tale istruzione risulta molto utile per permettere il Reset del sistema per mezzo di un altro sistema master, si pensi ad esempio ad una rete composta da vari nodi. Un altro utilizzo particolarmente utile potrebbe essere associato ad un sistema diagnostico, si pensi ad esempio che il software si sia reso conto che qualcosa è accaduto e si siano persi dei dati, l'istruzione Reset potrebbe tornare utile.



## Le porte d'ingresso uscita

Tutti i microcontrollori, per essere di qualche utilità, oltre ad eseguire delle istruzioni, devono poter essere in grado di interagire con il mondo esterno. In particolare devono poter leggere delle informazioni e/o scrivere dei risultati. Un microcontrollore che non ha nessuna possibilità di leggere delle informazioni esterne e non visualizza o pone nessun risultato come uscita, non ha nessuna ragione di essere programmato. Proprio per tale ragione, al fine di avere un'utilità, i PIC possiedono, oltre ai pin di alimentazione, molti altri pin che possono essere impostati per poter leggere o scrivere delle informazioni. In questo modo i PIC hanno una ragione per eseguire le istruzioni interne. In particolare ogni pin può essere in generale un pin digitale di I/O (Input/Output). Dal momento che il PIC possiede molte periferiche interne i pin oltre a poter essere utilizzati come I/O standard, sono spesso multiplexati con funzioni di altre periferiche, ovvero hanno più funzioni. I vari pin del PIC sono raggruppati in piccoli insiemi al massimo di 8 pin; ogni gruppo viene nominato PORTx, dove x prende il valore di una lettera A, B, C... la lettera massima dipende dal numero di gruppi presenti, ovvero dal numero di pin disponibili sul PIC. Non sempre il gruppo è formato da 8 pin, alcune volte la porta è incompleta, ovvero formata da meno di 8 pin. Il pin di ogni gruppo prende il nome della porta, seguito da un numero crescente compreso tra 0 e 7. In particolare i pin della PORTB saranno RB0..RB7. I raggruppamenti del PIC18F4550 sono visibili alla destra di Figura 3. In Figura 15 è riportato lo schema a blocchi di un pin generico.

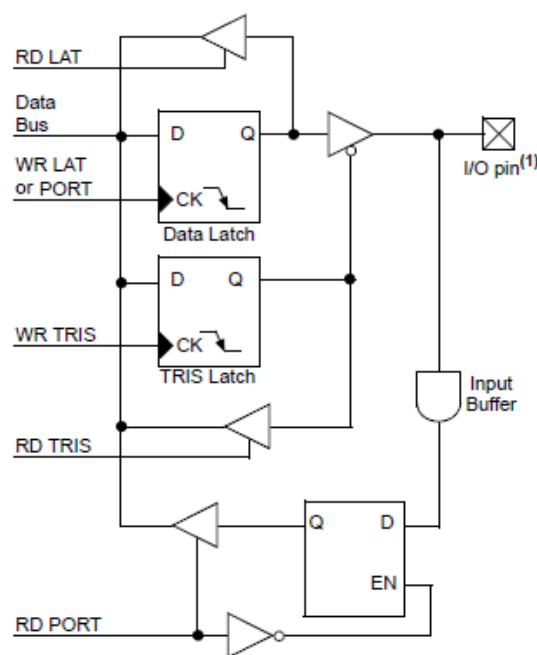


Figura 15: Schema a blocchi delle porte di uscita

Ad ogni gruppo di pin sono associati 3 registri, rispettivamente chiamati:

- **TRIS**

Il registro TRIS ha il compito di impostare la direzione (ingresso od uscita) del pin. Tale registro è composto da un byte ovvero da 8 bit; ad ogni bit corrisponde il pin della porta. Per esempio il bit 0 di TRISA imposta il pin RA0. Se il bit vale 0 il pin è impostato come output, mentre se vale 1, il pin viene impostato come ingresso. Una semplice regola per ricordare tale impostazione è quella di pensare 1 come I (Input) e 0 come O (Output).

---

- **PORT**

Il registro PORT viene utilizzato per scrivere un valore in uscita e/o leggere il valore del pin.

- **LAT**

Il registro LAT viene utilizzato per scrivere un valore in uscita e/o leggere il valore del latch<sup>33</sup> (non il pin). Per maggiori dettagli si legga di seguito.

Lo schema a blocchi di ogni pin, è riportato in Figura 15. In tale schema a blocchi non è visualizzato l'hardware specifico per ogni pin, necessario per gestire le opzioni di ulteriori periferiche. In particolare è visibile la presenza di due latch, uno è associato al registro TRIS, mentre il secondo è associato al registro PORT. Quando l'uscita Q del latch TRIS vale 1, il buffer che porta l'uscita Q del latch PORT all'uscita del pin, viene disabilitato, ovvero posto ad alta impedenza. Questo permetterà di poter leggere il pin senza essere influenzati dal valore del latch PORT. Il valore del registro TRIS normalmente viene impostato all'inizio del programma e non viene più cambiato, ma nulla vieta di sviluppare applicazioni in cui sia necessario cambiare il valore del registro TRIS durante l'esecuzione del programma. Qualora l'uscita Q del latch TRIS valga 0, viene abilitato il buffer di uscita, che permette di portare il valore dell'uscita Q del registro PORT in uscita al pin.

Vediamo ora meglio cosa sono i registri PORT e LAT, e quali sono le loro differenze. In particolare, per chi ha già studiato i PIC16, quale per esempio il famoso PIC16F84 e PIC16F877, il registro LAT è una cosa nuova, infatti i PIC citati possiedono solo il registro TRIS e PORT. Effettivamente è possibile scrivere 1 o 0 sul pin di uscita facendo solo uso del registro PORT; inoltre è anche possibile leggere il valore del pin di uscita e sapere se vale 0 o 1. Da quanto detto, non si capisce bene la ragione per cui nei PIC18 sia stato introdotto il registro LAT, per mezzo del quale è possibile a sua volta scrivere il pin di uscita o leggere il valore del latch di uscita (non il valore del pin). Il registro LAT è funzionalmente identico al registro PORT in scrittura, infatti in Figura 15 è possibile vedere che il data latch scrive il valore presente sul data bus in uscita, indistintamente se si effettua una scrittura sul registro PORT o LAT. La differenza dei due registri è nella fase di lettura, infatti quando si legge il registro PORT, il valore che viene posto sul data bus è proprio quello del pin, mentre quando si effettua la lettura del registro LAT, il dato posto sul data bus è il valore d'uscita del latch.

*Bene, allora qual'è la differenza?! Quando pongo il pin ad 1, l'uscita del latch vale 1 e il pin in uscita vale 1, dunque leggere il registro PORT o LAT è uguale!*

La differenza è proprio qui, il valore del registro PORT e LAT non sono sempre uguali! Il fatto va ricercato esternamente al PIC. Infatti la periferica o carico al quale il pin è collegato, potrebbe trovarsi ad una certa distanza dal PIC stesso, tale distanza potrebbe essere di soli 10cm, ma le piste potrebbero avere elementi parassiti (R, C ed L) tali per cui sia necessario del tempo prima che il valore cambi di stato da 0 a 1 o viceversa. Dunque il valore del pin potrebbe valere 0, quando invece il valore in uscita al latch è 1. Dunque leggere il pin tramite il registro PORT o leggere il latch per mezzo del registro LAT può avere le sue differenze<sup>34</sup>.

Questo causa dei problemi quando vengono eseguite istruzioni per cui, prima di cambiare il valore del bit è richiesto sapere il valore del registro LAT.

Dal momento che in scrittura i due registri LAT e PORT sono uguali, mentre in lettura

---

<sup>33</sup> Il latch rappresenta la cella elementare di memoria che permette di memorizzare un bit. Un registro è composto da un insieme di latch. Dunque una porta avrà fino a 8 latch.

<sup>34</sup> Il problema ora spiegato viene anche a dipendere dalla velocità (frequenza) a cui vengono letti e scritti i registri.

---

LAT può evitare dei problemi, *quando si utilizza un pin come uscita è consigliabile utilizzare sempre LAT sia in lettura che in scrittura*. Qualora il pin sia impostato come ingresso il pin andrà letto invece per mezzo del registro PORT. Vediamo ora in breve le peculiarità di ogni porta, ed in particolare le sue funzioni.

Quanto di seguito descritto fa riferimento al PIC18F4550, il numero di porte e funzioni può variare per PIC differenti. Per maggior dettagli fare sempre riferimento al datasheet del PIC utilizzato.

- **PORTA**

La PORTA possiede al massimo 7 pin, ovvero fino ad RA6, in particolare RA6 è disponibile solo se non si fa uso del quarzo esterno. Per poter utilizzare il pin RA6 sarà necessario, come si vedrà nei prossimi esempi, scrivere la direttiva:

```
#pragma config OSC = IRCIO67
```

Dal momento che si fa uso di una direttiva, tale impostazione non potrà essere più cambiata, se non riprogrammando il PIC. I pin della PORTA sono inoltre utilizzati come ingressi analogici. All'avvio del PIC tale funzione prevale sul valore impostato sul registro TRISA. Per poter utilizzare i pin della PORTA tutti come I/O è necessario scrivere 0x0F (ovvero 15) nel registro ADCON1<sup>35</sup>. Maggiori dettagli su tale registro sono riportati nel Capitolo dedicato agli ADC. Ogni volta che si avvia il PIC è bene che tra le prime istruzioni sia presente l'inizializzazione delle porte, in modo tale da evitare il danneggiamento dei sistemi collegati al PIC, nonché il PIC stesso. L'inizializzazione della PORTA, facendo uso di LATA invece di PORTA, è la seguente:

```
LATA = 0;           // Pongo a zero le uscite
ADCON1= 16;         // Imposto come I/O i pin della PORTA
TRISA = 0b00001111; // RA0..RA3 sono input, il resto è output
```

Il valore di ADCON1 e TRISA potrebbe essere differente a seconda delle esigenze. In particolare il formato numerico che può essere utilizzato può anche variare (binario, esadecimale, decimale). Come detto il nome dei pin è RA0..RA6, se però si fa uso del registro LAT, come si vedrà i nomi utilizzati sono LATA0...LATA6.

- **PORTB**

La PORTB, diversamente dalla PORTA possiede 8 pin, dunque possiede i pin RB0..RB7 e LATB0...LATB7 rispettivamente. Come le altre porte, ogni pin è bidirezionale ma può essere anche utilizzato per altre funzioni. In particolare, all'avvio del PIC, i pin RB0-RB4 sono impostati come ingressi analogici. Per poterli utilizzare come ingressi digitali è necessario utilizzare la direttiva:

```
#pragma config PBADEN = OFF
```

Dal momento che si fa uso di una direttiva, tale impostazione non potrà essere più cambiata, se non riprogrammando il PIC. Qualora si volessero utilizzare i pin come

---

<sup>35</sup> Si faccia riferimento al Capitolo sul modulo ADC, per maggior dettagli su come impostare tale registro.

---

ingressi analogici è necessario porre ad ON l'opzione PBDEN. Una caratteristica importante della PORTB che spesso viene trascurata, è la presenza di resistori di pull-up che possono essere abilitati per mezzo del bit RBPU del registro INTCON2. Tale funzione è particolarmente utile nel caso si faccia uso di pulsanti esterni, infatti permette di evitare l'utilizzo dei resistori di pull-up altrimenti necessari per fissare il livello logico del pulsante aperto<sup>36</sup>. Un'altra caratteristica della PORTB è quella delle interruzioni al variare dello stato logico, cioè viene generata un'interruzione se uno qualunque dei pin RB4-RB7 cambia di valore, ovvero da 0 ad 1 o viceversa. Si vedranno esempi su quanto descritto, nei prossimi Capitoli. La PORTB può essere inizializzata per mezzo delle seguenti istruzioni:

```
LATB = 0;           // Pongo a zero le uscite
ADCON1= xx;         // xx da impostare se PBDEN = ON

TRISB = 0b00001111; // RB0..RB3 sono input, il resto è output
```

Il valore di ADCON1 e TRISB potrebbe essere differente a seconda delle esigenze. In particolare il formato numerico che può essere utilizzato può anche variare (binario, esadecimale, decimale).

- **PORTC**

La PORTC possiede 7 pin ovvero RC0-RC6 e LATC0...LATC6 rispettivamente. Tra le periferiche principali che utilizzano tale porta si ricorda la EUSART e il modulo USB, il quale fa però utilizzo anche della PORTA. Di default, diversamente dalle PORTA e PORTB i pin sono digitali, dunque la sua inizializzazione è più semplice:

```
LATC = 0;           // Pongo a zero le uscite
TRISC = 0b00001111; // RC0..RC3 sono input, il resto è output
```

Il valore di TRISC potrebbe essere differente a seconda delle esigenze. In particolare il formato numerico che può essere utilizzato può anche variare (binario, esadecimale, decimale). Una periferica di particolare importanza che fa uso dei pin della PORTC è il modulo PWM, frequentemente utilizzato in applicazioni robotiche per il controllo della velocità dei motori. Altra applicazione potrebbe essere il controllo dell'intensità luminosa di una lampada. Maggiori dettagli verranno dati nel Capitolo dedicato al PWM.

- **PORTD**

La PORTD possiede 8 pin ovvero RD0-RD7 e LATD0...LATD7 rispettivamente. Tra le periferiche principali presenti su tale porta si ricorda la presenza dell'Enhanced PWM, ottimizzato per il controllo dei motori, grazie alla presenza di uscite complementari. La sua inizializzazione è tipo quella della PORTC:

---

<sup>36</sup> Se si leggesse il valore di un pulsante aperto, equivarrebbe a leggere un valore fluttuante che potrebbe essere sia 0 che 1 e variare col tempo. Il resistore di pull-up fissa ad 1 il valore del pulsante aperto. Come alternativa è anche possibile utilizzare resistori di pull-down che fissano il valore del pulsante aperto a 0, ma visto che è presente il resistore di pull-up interno al PIC, è bene far uso di quello.

---

```
LATD = 0;                // Pongo a zero le uscite
TRISD = 0b00001111;     //RD0..RD3 sono input, il resto è output
```

Il valore di TRISD potrebbe essere differente a seconda delle esigenze.

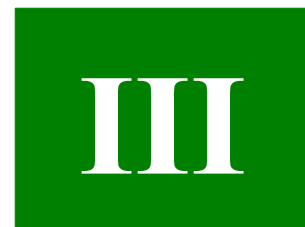
- **PORTE**

La PORTE possiede 4 pin ovvero RE0-RE3 e LATE0...LATE3 rispettivamente. Una loro peculiarità sta nel fatto che gli ingressi hanno un buffer a Trigger di Schmitt, permettendo agli ingressi di avere una maggiore immunità a disturbi esterni. Come per la PORTA la PORTE possiede i pin multiplexati con il convertitore analogico digitale. In particolare, dal momento che all'avvio del PIC PORTE è impostato con ingressi analogici, è necessario impostare propriamente il registro ADCON1:

```
LATE = 0;                // Pongo a zero le uscite
ADCON1= xx;              // xx da impostare se PBDEN = ON
TRISE = 0b00001111;     // RE0..RE3 sono input
```

Il valore di ADCON1 e TRISB potrebbe essere differente a seconda delle esigenze.

Dopo quanto esposto è bene mettere in evidenza che alcune periferiche, quando abilitate sovrascrivono il registro TRIS dunque è indifferente se il pin è stato impostato come ingresso o uscita. Altre volte è necessario che il pin sia propriamente impostato nel registro TRIS per permettere il corretto funzionamento della periferica. Quanto detto si applica come regola generale a tutte le porte. Per maggior dettagli è sempre bene far riferimento al datasheet del PIC utilizzato, in particolare si raccomanda la lettura del paragrafo associato alle porte e alla periferica d'interesse.



# Capitolo III

## Strumenti per iniziare

In questo Capitolo verranno elencati gli strumenti necessari per poter sviluppare le applicazioni spiegate in questo testo. In particolare verranno motivate le scelte che hanno condotto alla selezione di un determinato software/prodotto. Verranno inoltre brevemente illustrati i vari passi da seguire per installare la nostra tool chain (catena di strumenti).

### Perché si è scelto Microchip

Internet è pieno di materiale che spiega come realizzare sistemi con PIC, in particolare si trovano tutorial, progetti, programmatori e software per programmare. Molti dei tool sono gratuiti, ma il tempo e l'esperienza di partenza necessari per il loro utilizzo non sono trascurabili. Ci sono anche programmatori per PIC che richiedono un PIC già programmato dunque inutilizzabili come punti di partenza. Tutto il mondo gratuito per quanto bello possa essere ha una grave lacuna, la manutenibilità del tool. In particolare ogni volta che viene aggiunto un PIC o trovato un problema, la Microchip aggiorna i propri strumenti, mentre quelli gratuiti possono impiegare anni prima di aggiornarsi...se mai lo saranno!

Avere il servizio Microchip a portata di mano non è un beneficio da sottovalutare, specialmente se consideriamo il fatto che una delle politiche Microchip è quella di vendere i propri strumenti di sviluppo a basso costo, sviluppando strumenti per diverse esigenze, dall'hobbysta al professionista. La Microchip sviluppa inoltre anche il compilatore per i propri microcontrollori rendendo il tutto disponibile gratuitamente. In particolare i compilatori sono rilasciati gratuitamente sotto licenza gratuita per applicazioni accademiche. Il seguente testo rientrando in questa fascia d'interesse vi permetterà a pieno titolo di scaricare gratuitamente il compilatore C Academic Version anche noto C18.

Visitando il sito della Microchip troverete inoltre molte Application Note relative ai propri strumenti nonché dispositivi. Dunque, scegliendo gli strumenti Microchip o compatibili Microchip, come la scheda di sviluppo scelta, avrete la possibilità di avere sempre gli strumenti mantenuti dal produttore. Scegliendo invece di realizzare i vostri strumenti, sarete lasciati allo sbaraglio delle incertezze e delle difficoltà che incontrerete. Passerete giorni a cercare di capire il problema, per poi scoprire che l'hardware da voi usato non era supportato dal software. La spesa che sosterrete comprando un programmatore Microchip vi farà risparmiare tempo e anche denaro!

### L'ambiente di sviluppo MPLAB

Come prima cosa, prendendo per assodato che avete un PC con installato almeno XP, dovrete procurarvi il primo software gratuito offerto dalla Microchip, MPLAB. Tale software può essere scaricato dal sito della Microchip [www.microchip.com](http://www.microchip.com) dal menù *Design->Development Tool* dal gruppo *Software*. La versione trattata nel seguente testo è la MPLAB versione 8.40.

---

MPLAB è un IDE, acronimo inglese di Integrated Development Environment ovvero ambiente di sviluppo integrato. Per mezzo dell'IDE MPLAB è possibile creare progetti per ogni tipo di PIC, dai PIC10F ai PIC32 e scrivere programmi sia in C che in assembler. L'ambiente integra inoltre gli strumenti di sviluppo quali il programmatore, l'emulatore e il Debug sia hardware che software. Il Debug software può essere fatto tramite il simulatore incorporato. Tutto quanto quello descritto è 100% gratuito sia per applicazioni accademiche che commerciali. Avere un ambiente di sviluppo integrato permette di accelerare notevolmente i tempi di sviluppo, infatti dalla scrittura alla compilazione e alla successiva programmazione del dispositivo, bastano pochi click di mouse. Se si volesse si potrebbe anche evitare di utilizzare tale ambiente di sviluppo, visto che gran parte dei software Microchip sono tra loro integrati sfruttando semplice linea di comando. Personalmente non ho mai usato tale approccio e ve lo sconsiglio.

Se avete intrapreso la strada del costruirvi il programmatore da solo e vi volete fare ancora del male potete a vostro scapito utilizzare la linea di comando!

Come visto l'IDE scelto è della Microchip, ma sul mercato sono presenti anche altri IDE noti per la loro qualità. Uno dei migliori ambienti di sviluppo era quello della HI-TECH recentemente acquisita dalla Microchip. Un altro famoso ambiente di sviluppo completo di tutti gli strumenti di sviluppo, è quello sviluppato da mikroElektronika [www.mikroe.com](http://www.mikroe.com).

Da diversi forum ho potuto constatare che molti neofiti si trovano bene con tale ambiente di sviluppo. La società fornisce una versione scaricabile gratuitamente ma i suoi limiti possono creare qualche fastidio. Vi rimando al paragrafo precedente per le mie motivazioni sullo scegliere l'ambiente di sviluppo Microchip.

## Il compilatore C

Una volta installato l'ambiente MPLAB, è possibile installare il compilatore C. L'ambiente MPLAB permette di integrare diversi compilatori oltre a quello della Microchip noto come C18. Il più noto dei compilatori era quello della HI-TECH, che come detto è stata di recente acquisita dalla Microchip. Dalla sua acquisizione il compilatore C della HI-TECH è ancora supportato e scaricabile in una versione “Lite” per applicazioni non commerciali, direttamente dal sito della Microchip. Dal momento che Microchip ora supporta due compilatori, la situazione marketing è leggermente cambiata. Per tale ragione hanno deciso di cambiare il nome del compilatore C18 a “C for PIC18” rendendo il nome più generico e vicino al nome della HI-TECH. Stessa sorte hanno avuto, per medesima ragione gli altri compilatori dei PIC a 16 e 32 bit. Oltre a questo piccolo cambio di nome, la versione gratuita del compilatore, prima nota come C18 Student Version, è oggi chiamata Academic Version.

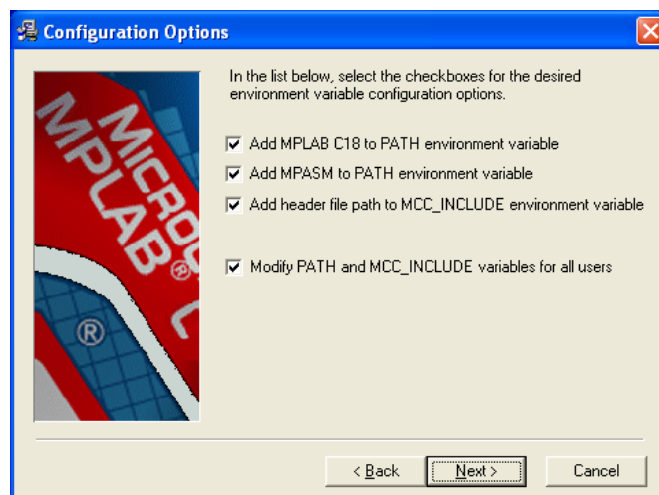
Il compilatore selezionato, ancora una volta, è quello originale della Microchip, ovvero il C per PIC18 ovvero C18. E' possibile scaricare la versione gratuita nominata Academic Version<sup>37</sup> dal sito della Microchip [www.microchip.com](http://www.microchip.com). La versione trattata nel seguente testo è la MPLAB-C18-Lite-v3\_34.

Una volta scaricato il file ed accettata la licenza, è possibile installare il tutto. In particolare si consiglia di selezionare le opzioni di inclusione dei percorsi di sistema utilizzati dall'ambiente di sviluppo e di aggiornare MPLAB, come riportato in Figura 16 e Figura 17.

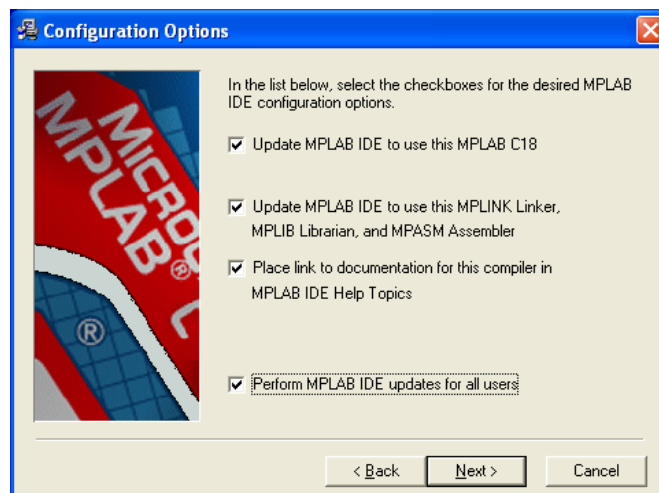
---

<sup>37</sup> La versione gratuita del compilatore C per PIC18 possiede alcuni limiti associati all'ottimizzazione. In particolare le ottimizzazioni del compilatore sono effettuate solo per i primi 30 giorni dall'installazione. Da un punto di vista delle applicazioni sviluppate in questo testo, tale limite non sarà nemmeno avvertito. Si ricorda che la versione free può essere utilizzata solo per applicazioni non commerciali.





**Figura 16:** *Impostazione delle variabili di sviluppo.*



**Figura 17:** *Impostazione per aggiornare MPLAB*

Il percorso standard d'installazione del compilatore C è all'interno della directory MCC18. Il percorso radice può essere variato a piacimento. All'interno della directory MCC18 sono presenti diverse sottocartelle. E' bene mettere subito in evidenza alcune cartelle che verranno di seguito utilizzate.

- **bin**

All'interno di questa cartella sono presenti i sorgenti del compilatore e del linker, utilizzati per creare il vostro codice da caricare all'interno del PIC. All'interno di tale directory, oltre ad altre applicazioni, è presente la sottodirectory LKR all'interno della quale sono contenuti i file utilizzati dal linker, con estensione .lkr. Per ogni PIC è presente un file che dovrà essere incluso nel progetto. Se per esempio stiamo utilizzando il PIC18F4550 bisognerà includere il file 18f4550\_g.lkr. Al suo interno sono presenti informazioni utilizzate dal linker relative all'organizzazione della memoria interna al PIC. Tale organizzazione varia a seconda delle opzioni utilizzate per la compilazione stessa. Il fatto di attivare o meno le funzioni di Debug, per esempio, cambierà l'organizzazione della memoria. Consiglio di aprire il file d'interesse almeno



---

una volta, il file è scritto in testo normale ed è facilmente comprensibile. Come si vedrà a breve, qualora si faccia uso del file linker standard, con le nuove versioni MPLAB non è più obbligatorio includere il file .lkr.

- **doc**

All'interno di tale directory è presente tutta la documentazione base utile per l'utilizzo del compilatore. In particolare è presente la guida d'utilizzo (hlpC18ug) e la guida per iniziare (MPLAB-C18-Getting-Started) utile per avere una buona visione delle funzionalità del compilatore. Oltre a tali documenti molto importante è il documento di riferimento delle librerie presenti a supporto della programmazione (hlpC18Lib). Molte periferiche hanno infatti delle librerie già scritte che possono semplificare molto il lavoro. In particolare la libreria si divide in:

- ✓ Software peripheral library
- ✓ General software library
- ✓ Math library

Ultimo documento particolarmente importante è hlpPIC18ConfigSet ovvero il file che descrive tutte le opzioni di configurazione che è possibile attivare, disattivare e selezionare per ogni PIC. Ogni PIC possiede infatti particolari configurazioni, per le quali è necessario sapere l'opportuno settaggio. Tale file risulta particolarmente comodo all'inizio della programmazione e verrà descritto in dettaglio nei prossimi Capitoli.

- **h**

All'interno di tale directory sono presenti tutti gli header file delle librerie che è possibile includere. Come detto le librerie e le loro funzioni sono descritte all'interno del documento hlpC18Lib. Tali file dovranno essere inclusi in ogni progetto, a seconda delle esigenze, per permettere l'utilizzo delle librerie stesse. Oltre ai file di libreria sono presenti gli header file con il nome dei pin di ogni PIC. Come visibile dal datasheet ogni PIC ha molti registri con nomi differenti, tali nomi vengono definiti all'interno degli header file del PIC relativo in modo da poter usare gli stessi nomi utilizzati nel datasheet. Se per esempio stiamo utilizzando il PIC18F4550 bisognerà includere il file p18f4550.h. Se apriamo il file troveremo al suo interno la definizione di molte strutture dati. Un esempio è:

```
extern volatile near unsigned char PORTD;
extern volatile near union {
    struct {
        unsigned RD0:1;
        unsigned RD1:1;
        unsigned RD2:1;
        unsigned RD3:1;
        unsigned RD4:1;
        unsigned RD5:1;
        unsigned RD6:1;
        unsigned RD7:1;
    };
    struct {
        unsigned SPP0:1;
        unsigned SPP1:1;
        unsigned SPP2:1;
        unsigned SPP3:1;
        unsigned SPP4:1;
        unsigned SPP5:1;
    };
};
```

```
    unsigned SPP6:1;  
    unsigned SPP7:1;  
};  
} PORTDbits;
```

In cui vengono definite PORTD e PORTDbits con relativi bit. La struttura PORTDbits può essere utilizzata per accedere i singoli bit del registro. In particolare ogni registro, come si vedrà, ha una struttura con il nome del registro stesso tipo REGISTRO e la struttura REGISTRObits per poter accedere i singoli bit. Per accedere il singolo bit bisogna interporre un punto tra il nome del registro e quello del bit. Per esempio per accedere il bit RD1 bisogna scrivere PORTDbits.RD1.

- **lib**

All'interno di tale directory sono presenti i codici binari, ovvero gli eseguibili associati ad ogni libreria, il cui header file è dichiarato all'interno della directory h. Questi file, come si vedrà non sono direttamente inclusi, il linker andrà alla ricerca di tali file sapendo che deve includerli. La conoscenza del doverli includere discende dal fatto che sta a noi includere l'header file d'interesse.

## Utilizzare il C o l'Assembler?

Giunti alla fine dell'installazione del compilatore C è giusto spendere due parole sull'alternativa al compilatore C rappresentato dall'assembly o Assembler. L'assembler è in relazione diretta con il linguaggio macchina ed è il linguaggio più vicino al microcontrollore. Per mezzo di tale linguaggio è possibile scrivere il codice meglio ottimizzato in assoluto. Allora perché non utilizzare direttamente l'assembler, il cui compilatore è per altro 100% gratuito anche per applicazioni commerciali?

Una delle ragioni principali per cui è bene non scrivere direttamente in assembler è legato al fatto che la leggibilità del codice è notevolmente inferiore, rendendo dunque la programmazione propensa a sbagli e problemi difficilmente rintracciabili. La leggibilità del codice è una delle pratiche della programmazione più importanti; un indice di qualità del software è la sua manutenibilità, ovvero la possibilità di aggiornare e rimettere mani sul software in un secondo momento. Si capisce che se il software non è facile da leggere rimetterci le mani diventa complicato. A peggiorare il tutto, immaginate che il software debba essere letto da un'altra persona...e magari l'altra persona siete proprio voi! Ringraziereste il programmatore se avesse potuto scrivere in maniera più chiara<sup>38</sup>. Maggiori informazioni sulle pratiche da utilizzare per scrivere un codice di buona qualità possono essere trovate nel documento "Programming Practices" scaricabile dal sito [www.laurtec.com/artorius](http://www.laurtec.com/artorius).

Un altro neo della programmazione in assembler è legata al fatto che il livello di conoscenza del microcontrollore che viene richiesta è più alta. Infatti quando si programma in C il livello di astrazione è più alto dunque è possibile leggere un codice C in maniera più simile ad un libro.

Nonostante abbia denigrato il linguaggio assembler è bene ora dargli il giusto riconoscimento. Diversamente da quanto ho più volte sentito in ambiente universitario che l'assembler non è più usato, questo viene ancora utilizzato molte volte e non solo per i PIC. In PIC con poca memoria flash non si può per esempio pensare di scrivere in C, è bene scrivere in assembler. Qualora si dovesse ottimizzare del codice è possibile integrare pezzi di codice assembler direttamente nel codice C (usando un'istruzione apposita).

---

<sup>38</sup> Un altro modo per migliorare la leggibilità del software è quello di usare buoni commenti e nomi di variabili appropriate.

---

Ottimizzare il codice ha senso solo dopo essersi resi conto che c'è realmente esigenza di ottimizzare del codice. Infatti l'ottimizzazione del codice porta in generale a una minor leggibilità del codice, dunque al fine di preservare la sua leggibilità è bene non ottimizzare fino a quando non è necessario.

Alcuni dati statistici hanno messo in evidenza che normalmente è il 20% del codice a prendere 80% delle risorse; questa analisi statistica è anche nota come regola 20 80. Quanto appena detto mette in evidenza che quando ci si rende conto che bisogna ottimizzare, non bisogna ottimizzare a caso ma bisogna andare alla ricerca di quel 20% di codice responsabile dei nostri problemi. Un'altra cosa importante da non dimenticare, è quella di verificare sempre i miglioramenti ottenuti quando si è ottimizzato un pezzo di codice. Se i benefici non sono quelli che ci si aspetta, è meglio togliere l'ottimizzazione e tenere il codice leggibile.

## Il programmatore

Ogni volta che si scrive un programma, sia esso in C o in assembly, per dar vita al programma è necessario poter installare o caricare il programma all'interno del microcontrollore. Il programma, come si vedrà nei prossimi Capitoli, per poter essere installato deve essere prima compilato. La fase di compilazione genera un file in linguaggio macchina, ovvero la traduzione del nostro programma C nel linguaggio del microcontrollore.

Sarà proprio questo file in linguaggio macchina che verrà caricato all'interno del microcontrollore. Per poter fare questo è necessario un programmatore, ovvero di un dispositivo al quale poter attaccare il nostro microcontrollore o scheda di sviluppo e che permetta di caricare il programma all'interno del PIC. In alcuni casi, se nel PIC è stato precedentemente caricato un bootloader, è possibile caricare il programma per mezzo di una normale connessione RS232 o USB.

In commercio è possibile trovare un'ampia gamma di programmatori per PIC ma tutti hanno lo svantaggio di non essere direttamente integrabili con l'ambiente IDE della Microchip e non supportano, se non con lunghi ritardi, gli ultimi modelli dei PIC. Per tale ragione anche in questo caso si consiglia un programmatore della Microchip.

Nel seguito del testo si farà uso del programmatore ICD 2 della Microchip, gentilmente regalatomi dalla stessa durante un training a Monaco. In ogni modo per completezza si descriveranno i più importanti programmatori e debugger offerti dalla Microchip per la fascia media professionale.

- **PICKIT 2**

E' il cavallo di battaglia tra i programmatori ma possiede tutte le funzioni richieste da un hobbysta o studente alle prime esperienze. E' integrabile nell'ambiente IDE MPLAB e programma i PIC10F, PIC12F5xx, PIC16F5xx, midrange PIC12F6xx, PIC16F, PIC18F, PIC24, dsPIC30, dsPIC33, e PIC32. Il programmatore può inoltre essere utilizzato come data logger per mezzo del programma Programmer Logic Analyzer scaricabile direttamente dalla home page del PICKIT 2. Il suo costo è veramente vantaggioso, viene venduto singolarmente o con piccola evaluation board con un PIC della famiglia PIC16. Sul mercato sono anche presenti dei cloni. Personalmente non li ho mai provati, ma visto che la microchip stessa ha rilasciato gli schemi del suo PICKIT 2, non escludo che non abbiano problemi.

- **PICKIT 3**

E' il fratello maggiore del PICKIT 2, il suo costo è circa il doppio, questo è dovuto al miglior supporto e capacità di debug.

---

- **ICD 2**

E' il primo programmatore e real-time debugger per una fascia di utilizzatori professionisti. Le capacità di Debug sono migliorate rispetto ai programmatori della serie PICKIT. Come quest'ultimi supporta tutta la serie di microcontrollori sviluppati dalla Microchip ed è integrabile nell'ambiente di sviluppo MPLAB.

- **ICD 3**

Nuovamente un fratello maggiore! ICD 3 è stato recentemente introdotto per sostituire l'ICD 2. Tanto è vero che la Microchip ha iniziato un programma di riciclo per ritirare gli ICD2 ed ottenere un coupon per uno sconto del 25% sull'acquisto dell'ICD 3. Le caratteristiche più importanti dell'ICD 3 sono un ulteriore miglioramento delle capacità di Debug e un aumento della velocità di programmazione. Diversamente dagli altri programmatori, l'ICD 3 possiede al suo interno un'FPGA al fine di supportare le alte velocità di programmazione. Per scoraggiare ulteriormente l'acquisto dell'ICD2, l'ICD 3 costa meno dell'ICD 2! Unico neo dei programmatori/debugger della serie ICD è che il loro costo è circa 5 volte superiore al PICKIT 2.

Caratteristica di tutti i programmatori ora descritti, è che sono tra loro compatibili da un punto di vista meccanico, ovvero il connettore e piedinatura dei vari programmatori è la stessa. Questo permette di sviluppare una scheda e poterla programmare con uno qualunque dei programmatori ora descritti. L'installazione del programmatore deve avvenire dopo l'installazione dell'ambiente di sviluppo MPLAB. E' indifferente se il C18 è stato già installato o meno. Per i passi da seguire si rimanda al manuale del programmatore selezionato. Si fa presente che alcuni programmatori hanno la possibilità di alimentare la scheda di sviluppo (denominata anche target). Una regola generale da seguire è che il programmatore deve essere collegato al PC e propriamente alimentato prima di essere collegato alla scheda target, che a seconda del programmatore dovrà o meno essere alimentata.

Nonostante si sia scoraggiato l'utilizzo di altri programmatori, è naturalmente possibile utilizzarne di altri, soprattutto se si pensa di utilizzare PIC standard per applicazioni hobbystiche. Come però già detto costruirsi un programmatore può portare la spesa a prezzi più alti che non un PICKIT 2, il quale ha tutti i vantaggi di un buon programmatore. Nel caso in cui, per una qualunque ragione si avesse un programmatore di altra marca è comunque possibile costruire un semplice connettore per adattare il connettore del proprio programmatore con quello della Microchip. I pin utilizzati per la programmazione, oltre a quelli di alimentazione, sono i seguenti:

**RB6:** Funzione PGD

**RB7:** Funzione PGC

**MCLR:** Tensione di programmazione

## Scheda di sviluppo e test

Questa volta la storia è un po' diversa e merita qualche nota. La scheda di sviluppo è in generale una scheda che permette di testare i vari programmi. Frequentemente la scheda può essere direttamente un proprio progetto dunque può essere di ogni genere. La Microchip come anche molti altri produttori forniscono molte schede più o meno ottimizzate per poter testare un determinato tipo di hardware, in generale se la scheda è flessibile per mezzo di espansioni, il suo costo lievita intorno a 150euro, più il costo delle varie schede di espansione.

Per la scheda di sviluppo, quello che realmente serve, è che abbia l'hardware che si vuole testare e che sia compatibile con gli strumenti Microchip, ovvero collegabile ai programmatori Microchip. Il connettore di programmazione come detto è standard, dunque una volta che si può utilizzare il PICKIT si può usare anche l'ICD<sup>39</sup>. Per tale ragione, se si volesse, si potrebbe anche realizzare su breadboard o su mille fori, una propria scheda di sviluppo. Personalmente, sconsiglio questa pratica poiché i fili volanti potrebbero causare qualche problema, ma non è certamente una via complicata. Nel testo farò uso della scheda che ho personalmente progettato ad hoc per scopi didattici e sulla quale si basa il corso stesso; la scheda in questione è nominata Freedom II ed è riportata in Figura 18.

La documentazione della scheda è scaricabile gratuitamente dal sito [www.LaurTec.com](http://www.LaurTec.com) alla sezione Progetti; si raccomanda la sua lettura prima dello studio dei progetti presentati nei Capitoli seguenti. Il PCB per la sua realizzazione è inoltre richiedibile, previa donazione di supporto, alla sezione Servizi. La donazione di supporto è utilizzata per realizzare opere e progetti gratuiti come il libro che state leggendo. Freedom II è particolarmente compatta e possiede un gran numero di periferiche ed è compatibile con i PIC 16F e 18F a 40 pin. Il connettore di espansione permette inoltre di collegare ulteriore hardware, permettendo praticamente di effettuare ogni tipo di esperimento. Naturalmente la scheda Freedom II è compatibile con i programmatori Microchip, permettendo un'integrazione completa con l'ambiente di sviluppo fin ora descritto. Altre schede di sviluppo possono essere utilizzate, ma i programmi che verranno descritti negli esempi dovranno essere modificati nel settaggio dei pin delle varie porte, in modo da poter utilizzare l'hardware d'interesse.



**Figura 18: Freedom II**

Come detto la scheda Freedom II è compatibile con la serie PIC16 e PIC18 a 40 pin. Nel seguente corso, si utilizzerà il PIC18F4550. Altri PIC potranno essere utilizzati, ma bisognerà accertarsi che i vari codici presentati siano opportunamente modificati.

<sup>39</sup> La Microchip per i programmatori utilizza due tipi di connettori, per i quali fornisce però l'adattatore stesso.

# Capitolo IV

## Salutiamo il mondo per iniziare

Non credo che ci sia nessun corso di un linguaggio di programmazione che non inizi salutando il mondo. Sarà una forma di educazione o per semplicità che anche noi inizieremo con un bel saluto. In questo Capitolo anche se non si capirà molto della programmazione, si illustrano i vari passi che portano alla progettazione e sviluppo di un'applicazione completa. Inizieremo illustrando come creare un nuovo progetto per poi compilarlo e caricarlo sulla scheda di sviluppo Freedom II.

### Il nostro primo progetto

Per poter iniziare a sviluppare una nuova applicazione, ovvero software da poter caricare all'interno del nostro microcontrollore, è necessario creare un progetto. Un progetto non è altro che una raccolta di file, in particolare sarà sempre presente il programma sorgente e file di libreria. L'ambiente di lavoro aggiungerà poi altri file per mantenere anche altre informazioni, ma di questi si parlerà in seguito.

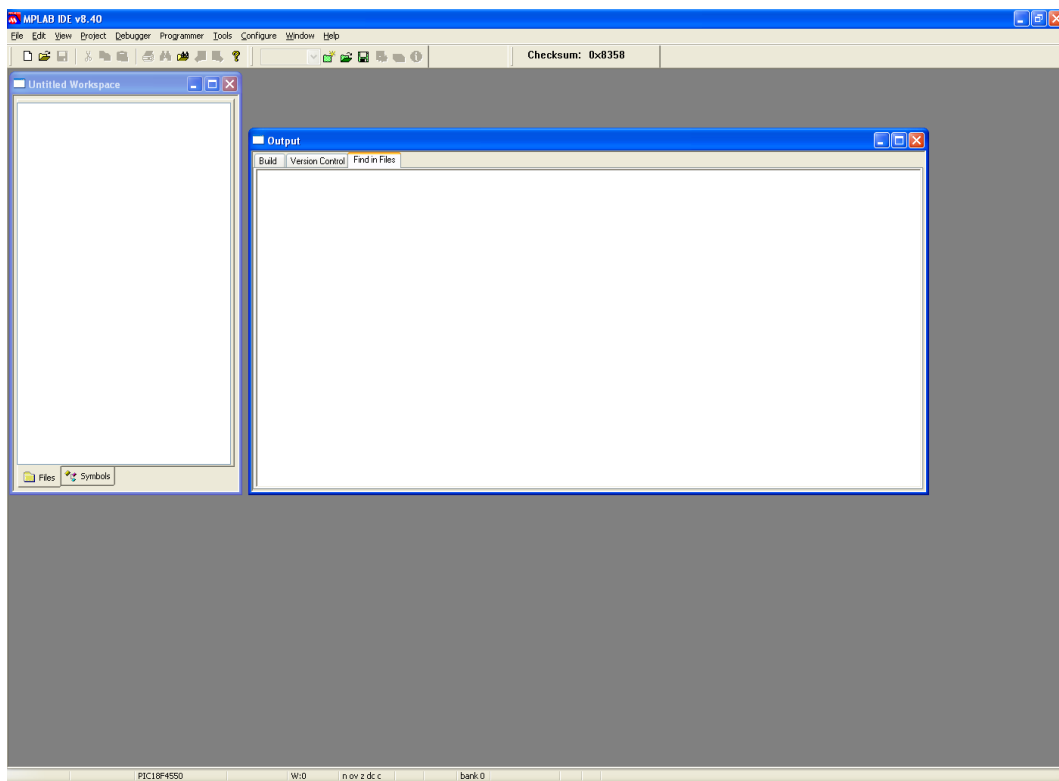
...allora si inizia.

Come prima cosa si deve avviare MPLAB, ovvero l'ambiente di sviluppo dal quale compieremo tutte le operazioni di sviluppo, programmazione, compilazione e simulazione. La schermata di avvio di MPLAB sarà, al suo primo avvio simile a Figura 19. Per creare un nuovo progetto bisogna andare sul menù *Project* e selezionare *Project wizard*.

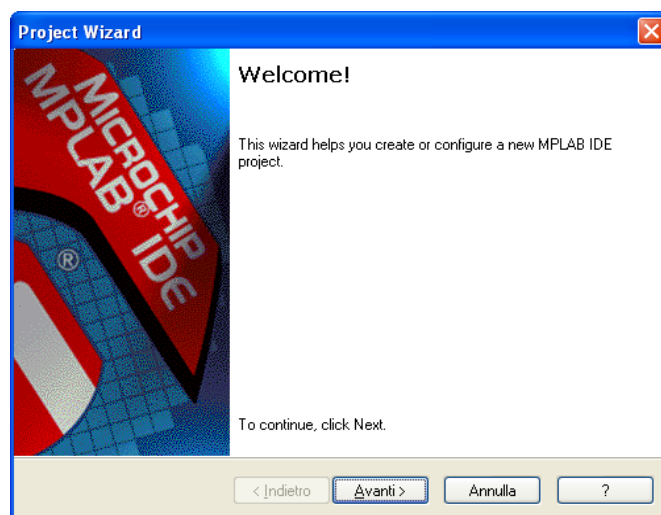
Si aprirà una finestra che guiderà la creazione di un nuovo progetto, come riportato in Figura 20. Cliccando sul pulsante *Avanti* si avrà la nuova finestra di Figura 21, dove bisognerà selezionare il PIC della famiglia PIC18 che si vuole utilizzare. Negli esempi che seguiranno si farà uso del PIC18F4550 che andrà dunque montato sul sistema embedded Freedom II<sup>40</sup>. In generale anche altri PIC potranno essere utilizzati senza alcun problema, ma si dovrà avere l'accortezza di cambiare l'header file, di cui si parlerà a breve, associato al PIC utilizzato. Dopo aver selezionato il PIC si può premere nuovamente il pulsante *Avanti*.

A questo punto, se la Tool Suite C18, ovvero il compilatore C18 è stato propriamente installato, comparirà la finestra di dialogo di Figura 22, con selezionato la voce Microchip C18 Tool suite. In particolare si fa presente che l'ambiente di sviluppo MPLAB supporta l'integrazione di altre suite di sviluppo; è infatti possibile installare compilatori per altre famiglie di PIC o ancora compilatori di altre marche. Una volta impostato la Tool Suite è possibile premere nuovamente *Avanti*.

<sup>40</sup> Un qualunque altro sistema di sviluppo o PIC della famiglia PIC18 è in generale utilizzabile.

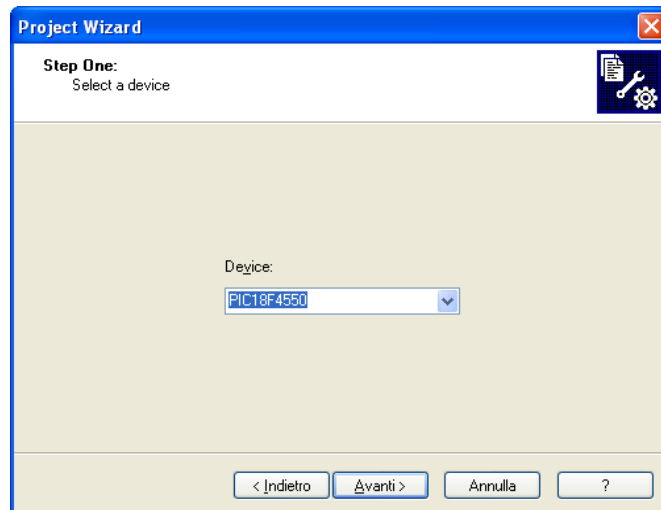


**Figura 19:** *Primo avvio di MPLAB*

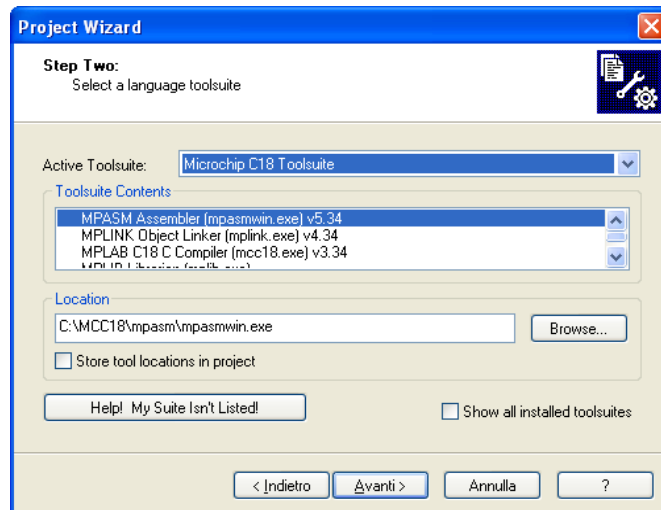


**Figura 20:** *Primo passo per la creazione del progetto*





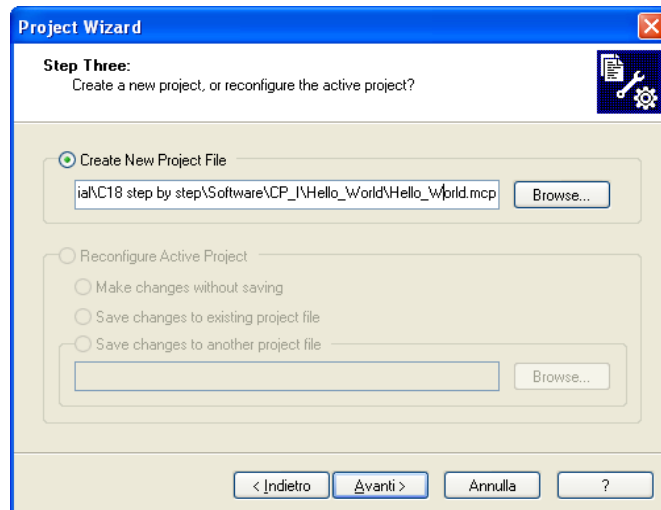
**Figura 21:** Finestra di dialogo per la selezione del PIC



**Figura 22:** Selezione della Tool Suite

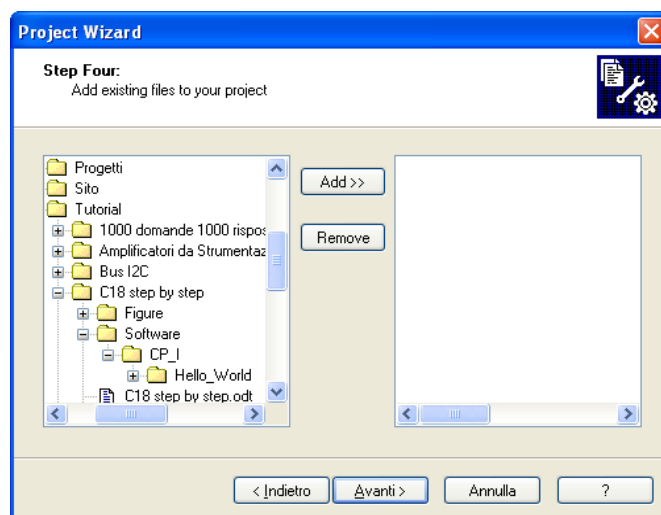
Una volta impostati i parametri principali associati all'Hardware da utilizzare viene visualizzata la finestra di dialogo di Figura 23, per mezzo della quale è possibile scrivere il nome del nostro progetto. Il nome del progetto creerà il file di progetto all'interno del percorso scelto. Si fa presente che è buona pratica avere una directory all'interno della quale vengono creati i vari progetti d'interesse. In particolare è bene nominare la directory che conterrà il nostro progetto con lo stesso nome del progetto stesso. Il progetto dell'esempio, come anche tutti gli altri progetti presentati sul testo possono essere scaricati gratuitamente dal sito [www.LaurTec.com](http://www.LaurTec.com).





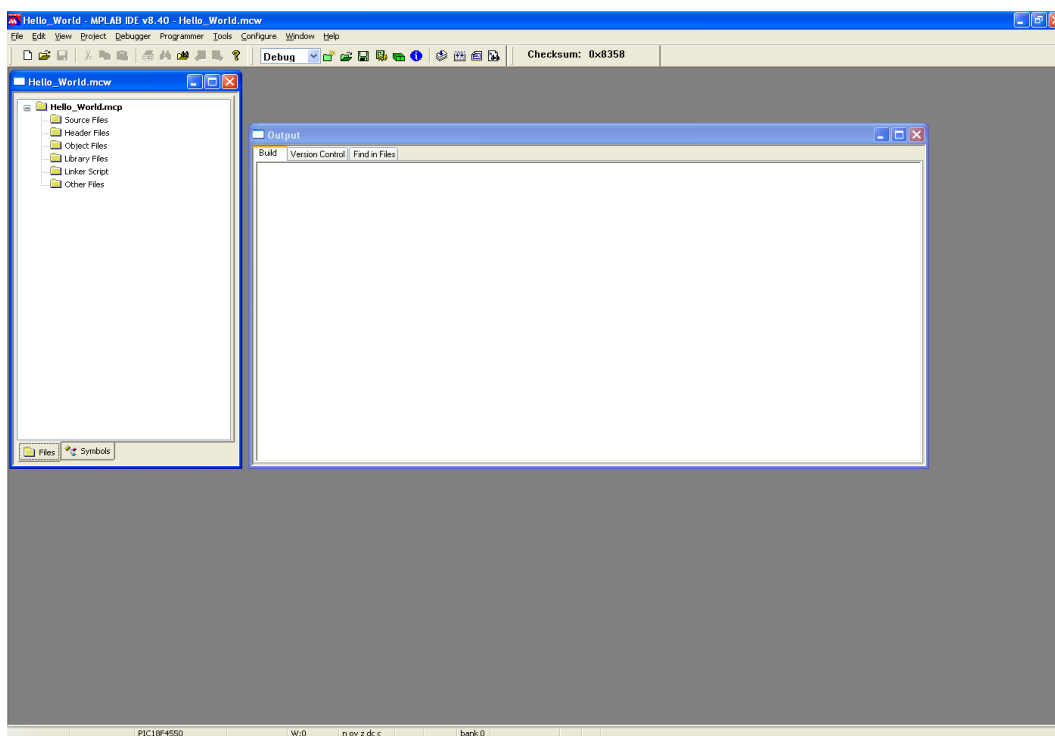
**Figura 23:** Impostazione del nome del progetto

Una volta impostato il nome del progetto è possibile premere il tasto *Avanti*. La nuova finestra di dialogo, mostrata in Figura 24 permette di aggiungere i file al nostro progetto. Tale pratica può ritornare utile se vi sono file sorgenti già scritti che si vuole includere nel progetto o per i quali si sta creando tale progetto. Come si vedrà a breve anche i file di .lkr e gli header file potrebbero essere inclusi a questo punto del progetto. In ogni modo in questo momento non si includerà nulla. I file d'interesse verranno inclusi direttamente dalla finestra del progetto, mostrando così il secondo modo per includere il file.



**Figura 24:** Impostazione dei file del progetto

Una volta premuto *Avanti*, verrà visualizzato un riassunto delle impostazioni del progetto, premendo nuovamente *Avanti*, comparirà nuovamente il nostro ambiente di sviluppo, all'interno del quale sarà però possibile vedere che è stato creato il nostro progetto Hello\_World, come riportato in Figura 25.



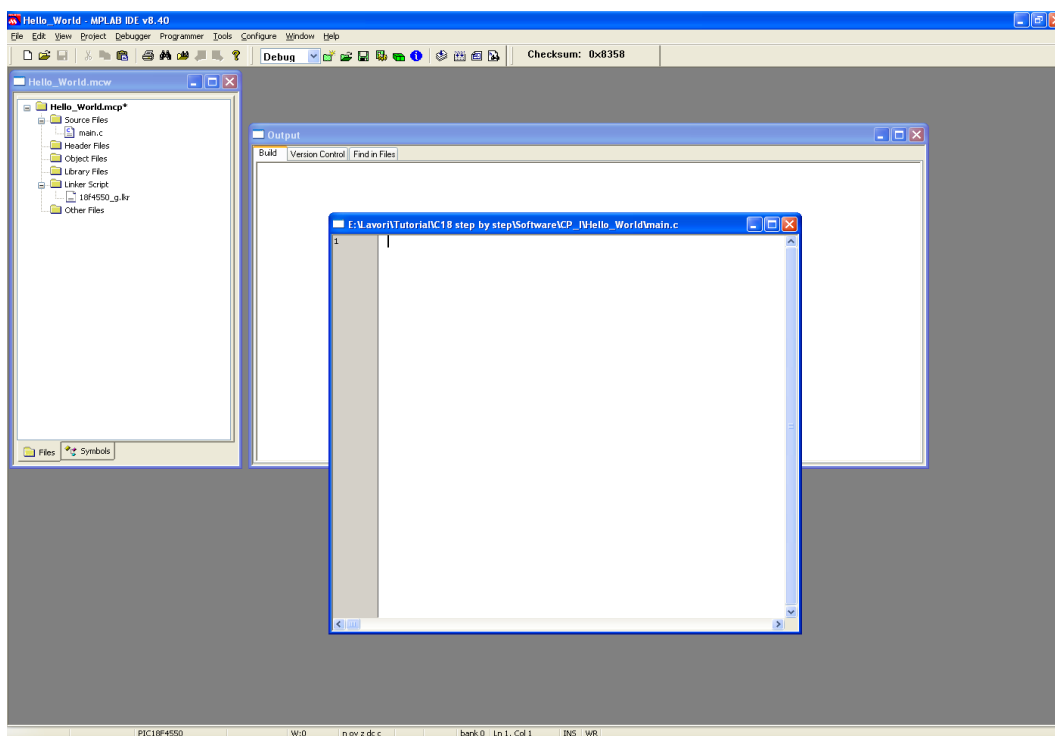
**Figura 25:** Ambiente di sviluppo al termine della creazione del progetto

A questo punto bisognerà aggiungere i file che non si sono inseriti precedentemente. Un file che è necessario aggiungere è il file per il linker con estensione .lkr. Come detto ogni PIC possiede più file di .lkr. Per inserire questo file bisogna selezionare la cartella Linker Script dalla finestra all'angolo sinistro della finestra principale e premere il tasto destro del mouse. Fatto questo bisogna selezionare la voce *Add File...* Qualora la finestra di dialogo non si apra direttamente nella cartella in cui sono presenti i file .lkr, dalla directory d'installazione del compilatore, andare al seguente percorso [...]MCC18/bin/LKR e cercare il file 18f4550\_g.lkr. Aggiunto il nostro file possiamo ora aggiungere il file principale dove scrivere il nostro programma. In C è tipico scrivere il programma principale all'interno di un file nominato main.c dove main significa principale<sup>41</sup>. Per inserire il nostro file bisogna creare una nuova finestra di testo dal menù *File* → *New*. Il file di testo così creato non appartiene però ancora al nostro progetto. Per poter integrare il file al progetto bisogna salvare il file da qualche parte e poi aggiungerlo alla cartella *Source Files*, come si è fatto per il file precedente. Ragionevolmente il file va salvato all'interno della directory in cui stiamo lavorando, ovvero [...]Hello\_World in modo da avere tutti i file in un solo punto. Dopo aver salvato il file di testo con il nome main.c<sup>42</sup> inserirlo nella cartella *Source Files* allo stesso modo con cui si è inserito il file .lkr. Al termine di queste operazioni il nostro ambiente di sviluppo avrà il nuovo aspetto riportato in Figura 26.

A questo punto, anche se apparentemente sembra tutto finito, rimane in realtà un altro passo, che se saltato può portare ad errori fastidiosi, soprattutto all'inizio della nostra esperienza. Quello che manca da fare è controllare che i percorsi delle librerie, header file e link siano propriamente impostati. Se così non fosse il compilatore, quando lo si avvierà non riuscirà a trovare le librerie, ed in articolare il linker non riuscirà a portare a termine la creazione del codice macchina finale. Ma niente paura, le impostazioni sono molto semplici.

<sup>41</sup> Altra abitudine è anche chiamare il file principale con un nome che descriva l'applicazione, quindi in questo caso potrebbe essere chiamato Hello\_World.

<sup>42</sup> Si fa notare che file sorgente in C deve avere estensione .c.



**Figura 26:** Ambiente di sviluppo al termine dell'aggiunta dei file

Per impostare i percorsi è necessario premere la piccola icona verde con l'ingranaggio giallo presente sulla Tool bar, ovvero le *Build Options*<sup>43</sup>. Si aprirà la finestra di dialogo come Figura 27. Dal Tab *Directory*, impostare i vari percorsi selezionabili dal menù a tendina *Show Directories for*. Accertarsi che i seguenti percorsi siano propriamente presenti e/o aggiunti (premere Applica all'aggiunta di ogni percorso):

*Include Search Path:* C:\MCC18\h  
*Library Search Path:* C:\MCC18\lib  
*Linker-Script Search Path:* C:\MCC18\bin\LKR

In particolare tali percorsi sono quelli standard per il compilatore C18. Il percorso assoluto potrebbe essere diverso a seconda del percorso d'installazione che si è scelto. Premendo il tasto *Suite Default* non accadrà nulla<sup>44</sup>, ma tra poco vedremo cosa fare. Nei prossimi Capitoli, quando si inizieranno ad utilizzare anche altre librerie, sarà necessario inserire i nuovi percorsi oltre a questi che dovranno sempre essere presenti. Il nostro progetto è ora terminato ed è possibile iniziare a scrivere il nostro programma.

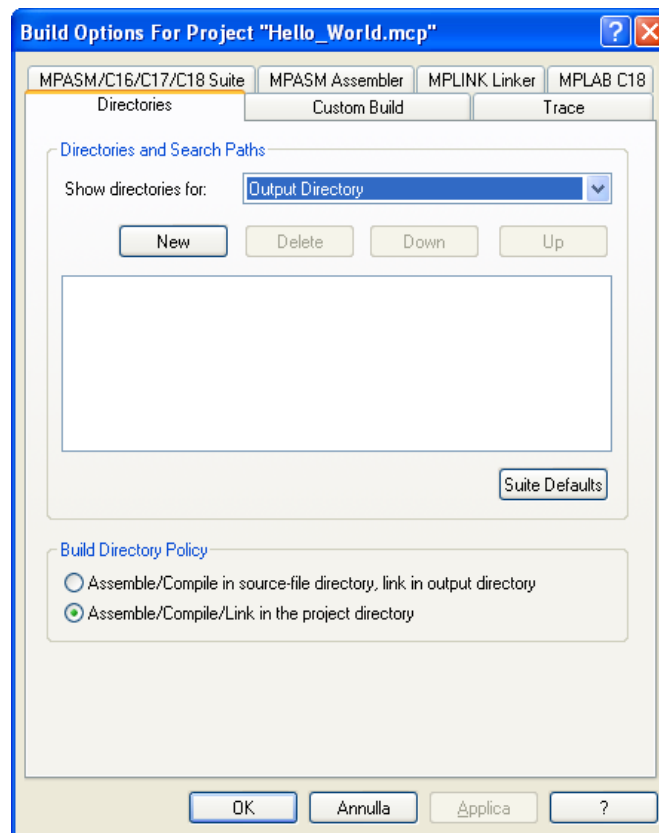
Quanto appena svolto per includere il file di linker e le librerie è in un certo modo uno strazio se dovesse essere ripetuto per ogni progetto; questo può essere evitato, vediamo come.

Nelle più recenti versioni di MPLAB e C18, ed in particolare in quella descritta, non è più necessario includere il file .lkr. Quello standard viene infatti automaticamente incluso qualora non venisse incluso manualmente. Ciononostante è richiesto includere il file .lkr qualora venissero apportate delle modifiche sulla struttura e gestione della memoria interna al PIC. In qual caso si dovrà includere il file .lkr modificato. Si capisce che è buona norma tenere il

<sup>43</sup> E' possibile aprire la stessa finestra di dialogo andando al menù *Project* → *Buil Options...* → *Project*.

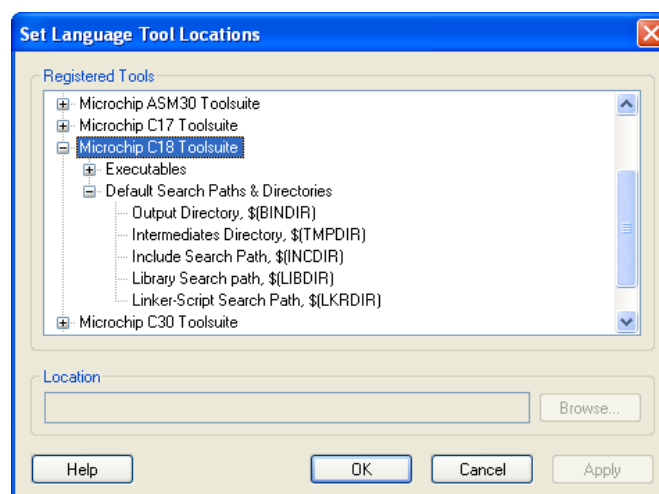
<sup>44</sup> Personalmente non apprezzo tale comportamento, visto che l'installazione avrebbe dovuto aggiungere i file alla variabile PATH.

file .lkr originale nel percorso standard e creare una copia da modificare in un altro percorso personale, dal quale poi includere il file.



**Figura 27:** *Impostazione dei percorsi*

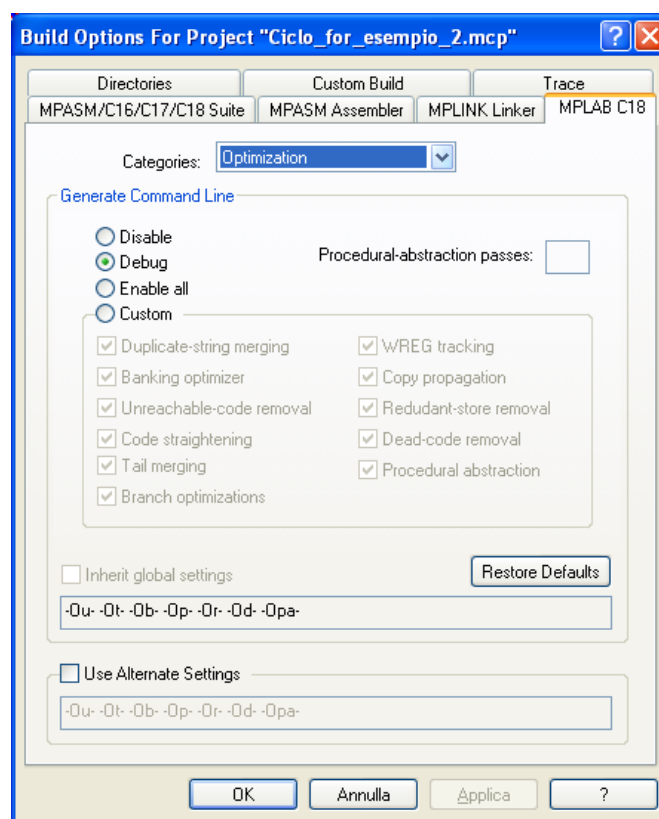
Per impostare i percorsi standard (non quelli personali) su dove cercare il file di linker, header file e librerie, è possibile procedere nel modo seguente: andare al menù *Project* → *Set language Tool Location* e selezionare *Mirochip C18 Toolsuite* come riportato in Figura 28.



**Figura 28:** *Impostazione dei percorsi standard*

A questo punto è possibile selezionare i singoli percorsi precedentemente discussi e aggiungerli per mezzo della funzione *Browse*. Una volta aggiunti, quando si creerà un nuovo progetto sarà possibile vedere i percorsi standard già nella nostra finestra di Figura 27. Qualora non fossero presenti basterà premere il tasto *Suite Defaults...* che adesso funzionerà, ed i percorsi standard verranno aggiunti in un solo colpo.

Un'altra impostazione importante associata ad un progetto è relativa al livello e tipologie di ottimizzazioni che è possibile abilitare. Le Opzioni di ottimizzazione possono essere selezionate dalla finestra di dialogo *Build Options*, selezionando il Tab *MPLAB C18* e selezionando la categoria *Optimization*, come visibile in Figura 29. Nel seguente testo si farà uso dell'opzione *Debug*, la quale disabilita tutte quelle ottimizzazioni che possono influenzare il normale flusso di esecuzione del programma.




**Figura 29:** Impostazione dell'ottimizzazione

Nel testo anche se verranno messi in evidenza alcuni esempi in assembler, l'argomento delle ottimizzazioni non viene trattato in maniera approfondita. Si fa presente che a seconda delle opzioni selezionate, della versione del compilatore e del compilatore stesso, il codice generato può essere differente dagli esempi che verranno mostrati. Gli esempi hanno infatti il solo scopo di mettere in guardia il lettore che vuole ottimizzare il codice facendo uso di strutture apparentemente più snelle. Qualora si abbiano esigenze particolari e problemi di spazio o di velocità di esecuzione, permettere al compilatore di ottimizzare può migliorare i risultati senza apportare variazioni al codice. I risultati vanno però testati per ogni singolo caso.

---

## Scrivere e Compilare un progetto

Una volta creato il progetto è possibile cominciare a scrivere il programma all'interno della finestra di testo che era stata nominata `main.c`. Normalmente questa fase, anche se in questo caso consisterà semplicemente in un copia ed incolla, dovrà essere preceduta dalla fase di analisi per la risoluzione del progetto. Non bisogna mai iniziare a programmare senza aver fatto un'opportuna analisi del problema. La fase di programmazione deve essere solo una traduzione della soluzione dal linguaggio umano in linguaggio C. In ogni modo questa non sarà la sede in cui vi insegnerò l'approccio alla programmazione, ma non posso non consigliarvi la lettura di testi per l'ingegneria del software.

Allora è giunto il momento di programmare...copiate ed incollate brutalmente il seguente codice facendo attenzione che non compaiano simboli strani. Quando sarete più esperti scriverete il vostro programma, ma in questa fase ho solo interesse di mostrare i passi da compiere per scrivere e compilare il file sorgente, ovvero il file C. Il programma non è molto complesso, ma dal momento che ogni sua linea è una cosa nuova, verrà analizzato in dettaglio, ma tempo al tempo. Una volta scritto o copiato il programma nella casella di testo, salvare il file e avviare la compilazione per mezzo della voce del menù *Project* → *Build All*, o l'icona sulla Tool bar .

Un altro modo per avviare la compilazione consiste semplicemente nel premere `Control+F10`, a voi la scelta.

```
#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS           Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

void main (void) {

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutti ingressi e RD0 come uscita
    LATD = 0x00;
    TRISD = 0b11111110;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    LATDbits.LATD0 = 1;
```

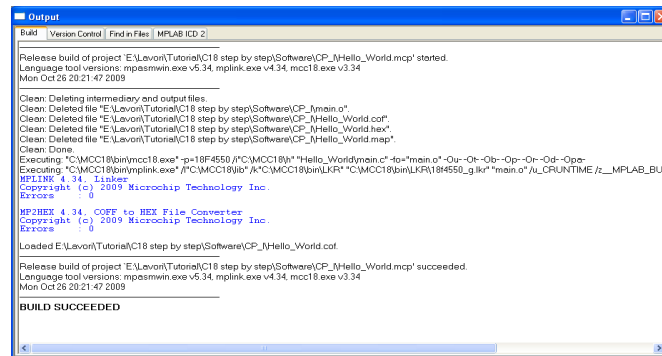
```

// Ciclo infinito
while (1) {

}
}

```

Quando si avvia la programmazione verrà presentata la finestra di dialogo riportata in Figura 30.



**Figura 30:** Impostazione Finestra di dialogo a termine compilazione

Nella finestra di dialogo, se tutto è andato a buon fine verrà visualizzato il messaggio BUILD SUCCEEDED, ovvero la compilazione ha avuto successo. In caso contrario verranno visualizzati gli errori e le warning che il programma ha generato. In particolare se sono presenti errori non verrà generato il codice macchina ovvero .hex per cui stiamo facendo tanti sforzi. Ogni messaggio di errore e warning è accompagnato anche dalla riga alla quale è stato generato il messaggio<sup>45</sup>. Per eliminare gli errori è bene procedere risolvendo il primo errore, infatti può capitare che altri errori siano solo dovuti ad effetti collaterali del primo. Per tale ragione, quando si pensa di aver risolto il primo errore, è bene ricompilare il codice. Questo dovrà essere ripetuto fino alla eliminazione di ogni errore.

Qualora dovessero essere generate delle warning il nostro file .hex verrà generato senza problemi. A questo punto è bene mettere in evidenza che ogni programma, per buona pratica di programmazione e robustezza delle nostre applicazioni, deve essere compilato senza nessun messaggio di warning. Qualora dovessero essere visualizzati dei messaggi di warning è bene capire fino in fondo il significato del messaggio e fare in modo che non venga visualizzato<sup>46</sup>. Le warning possono nascondere infatti dei bug subdoli...la cui caccia può essere molto poco divertente.

Dopo aver compilato il nostro primo programma, diamo un'occhiata all'interno della directory Hello\_World per vedere i vari file che sono stati generati per il nostro progetto.

### Hello\_World.mcp

Questo file rappresenta il file di progetto. Viene creato nel momento in cui viene creato un

<sup>45</sup> Frequentemente quando ci si scorda il ; alla fine di ogni riga, l'errore viene segnalato alla riga successiva. Dunque per trovare un errore è sempre bene guardare anche le righe che precedono la riga dove è stato segnalato l'errore.

<sup>46</sup> Un modo per far scomparire tutti i messaggi di warning è quello di disabilitarle per mezzo del menù *Build Options*, Tab *MPLAB C18*, sezione *Diagnostic Level*. Normalmente vengono visualizzati sia gli errori che le warning ma può essere scelto di visualizzare solo gli errori. Questa pratica è però altamente sconsigliata, le warning devono essere eliminate capendo la causa che le ha generate.

---

nuovo progetto. Per mezzo di tale file, facendo un doppio click sull'icona, è possibile direttamente riaprire il progetto salvato.

### **Hello\_World.mcw**

Questo file è molto simile al precedente, ma contiene le informazioni associate al Workspace. Da un punto di vista pratico ha le stesse funzioni del file precedente, infatti un doppio click sull'icona farà aprire il progetto precedentemente salvato.

### **Hello\_World.mcs**

Questo file viene creato dopo la creazione del progetto, contiene informazioni associate alle impostazioni e ai file del progetto stesso. Viene aggiornato al variare del progetto.

### **main.c**

Questo rappresenta il nostro file sorgente. Si ricorda che il nome potrebbe essere anche diverso ma la sua estensione deve essere .c. Un progetto potrebbe anche contenere più file sorgenti, ma solo uno conterrà, come si vedrà a breve, la funzione main.

### **Hello\_World.o**

Questo file rappresenta il file oggetto creato dal compilatore una volta compilato il nostro progetto. A seconda del numero dei sorgenti e progetti tra loro collegati, potranno essere presenti più file oggetti.

### **Hello\_World.hex**

Questo rappresenta il file in linguaggio macchina, scritto in esadecimale. Viene ottenuto per mezzo del linker, il cui compito è quello di legare, connettere tutti i file oggetto e librerie, rispettando le informazioni del file .lkr. Il suo formato è generalmente basato sullo standard Intel HEX ovvero semplice file ASCII leggibile con normale editor. Tale formato è accettato praticamente da tutti i programmatori e viene utilizzato anche come formato per scrivere i dati all'interno di memorie EEPROM, OTP, Flash e via dicendo. Nel nostro primo esempio il file di uscita è il seguente:

```
:0200000040000FA
:0600000063EF00F01200A6
:020006000000F8
:08000800060EF66E000EF76E05
:10001000000EF86E00010900F550656F0900F550FB
:10002000666F03E1656701D03DD00900F550606F50
:100030000900F550616F0900F550626F0900090071
:10004000F550E96E0900F550EA6E09000900090053
:10005000F550636F0900F550646F09000900F6CF91
:1000600067F0F7CF68F0F8CF69F060C0F6FF61C0C5
:10007000F7FF62C0F8FF0001635302E1645307E039
:100080000900F550EE6E6307F8E26407F9D767C020
:10009000F6FF68C0F7FF69C0F8FF00016507000EB2
:0600A000665BBFD71200F1
:0A00A600000EF36E00EE00F0040EF1
:1000B00001D81200EA6002D0EE6AFCD7F350E96082
:0600C0001200EE6AFCD7FD
:0A00C60013EE00F023EE00F0F86ADC
:1000D000019C04EC00F07FEC00F071EC00F0FBD729
:0200E00012000C
:0E00E200896A92688A6A93688B6A94688C6A4D
:0E00F000FE0E956E8D6A96688C80FFD712000A
:0200FE001200EE
:0200000040030CA
```



```
:010001000CF2
:010003001EDE
:010005008179
:010006008178
:00000001FF
```

Maggiori informazioni sul formato Intel HEX possono essere trovati nell'enciclopedia Wikipedia all'indirizzo [http://en.wikipedia.org/wiki/Intel\\_HEX](http://en.wikipedia.org/wiki/Intel_HEX).

### Hello\_World.map

Questo file possiede al suo interno le informazioni relative alle varie variabili e registri utilizzati nel programma. Viene generato come file di output dal linker. Al suo interno è possibile anche trovare l'informazione associata alla quantità di memoria utilizzata. Un modo più pratico per vedere la memoria utilizzata è per mezzo dello strumento *Memory Usage Gauge*. E' possibile eseguire tale strumento dal menù *View* → *Memory Usage Gauge*. Lo strumento visualizzato è riportato in Figura 31.

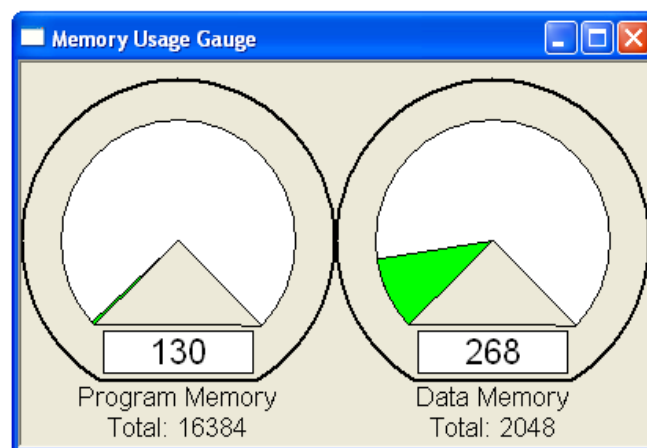


Figura 31: *Memory Usage Gauge*

Oltre ai file ora descritti possono essere generati anche altri file, in particolare dipende dalle impostazioni di compilazione. Qualora dovessero essere presenti degli errori verrà anche generato il file `nome_file.err` con la lista degli errori e dei messaggi di warning. Per maggior informazioni si faccia riferimento alla documentazione ufficiale sul compilatore e il linker,

## Programmare il microcontrollore

Una volta scritto il nostro programma e compilato con successo, è possibile scrivere il nostro file `.hex` all'interno del nostro PIC. Questa fase può essere differente a seconda del programmatore utilizzato. Nel caso di programmatori ufficiali Microchip, supportati dall'ambiente MPLAB sarà necessario selezionare il programmatore che si ha a disposizione all'interno del menù *Programmer* → *Select Programmer*. I passi che seguiranno faranno riferimento al programmatore ICD2. La selezione del programmatore farà comparire la tool bar di Figura 32.



**Figura 32:** Tool bar del programmatore disattiva

A questo punto è possibile, se non precedentemente fatto, collegare il programmatore. Si ricorda che il programmatore ICD2 come anche altri programmatori deve essere collegato all'USB (dunque essere alimentato) prima di essere collegato alla scheda di sviluppo (*Target Board*). Una volta collegato il programmatore al PC, è possibile collegarlo alla scheda di sviluppo, la quale deve essere già alimentata. Fatto questo è possibile premerne l'unica icona attiva della Tool bar, la quale permette di attivare il programmatore. La Tool bar verrà modificata come riportato in Figura 33.



**Figura 33:** Tool bar del programmatore attiva

Nel caso in cui si attivi il programmatore senza che sia collegato alla scheda di sviluppo, si avrà il seguente errore.

*Connecting to MPLAB ICD 2*

*...Connected*

*Setting Vdd source to target*

*ICDWarn0020: Invalid target device id (expected=0x90, read=0x0)*

*Running ICD Self Test*













*... Failed Self Test. See ICD2 Settings (Programmer->Settings) (status tab) for details.*

*MPLAB ICD 2 ready for next operation*

Questo errore e warning viene anche segnalato se la scheda di sviluppo non è propriamente alimentata, ovvero il programmatore non riesce a riconoscere il PIC sulla board, il quale dovrà essere lo stesso selezionato per il progetto.

Dal momento che si sta utilizzando un sistema di sviluppo generico, prima di avviare la programmazione è bene accertarsi che la scheda è propriamente impostata per l'applicazione che si andrà a scrivere sul PIC. L'impostazione è sempre bene farla con la scheda di sviluppo spenta. Nel caso del nostro primo programma l'impostazione dei Jumper della scheda di sviluppo Freedom II deve essere come riportato in Figura 34<sup>47</sup>:

<sup>47</sup> Maggiori informazioni sulle impostazioni della scheda di sviluppo Freedom II possono essere trovate sulla documentazione ufficiale della scheda, scaricabili dal sito [www.LaurTec.com](http://www.LaurTec.com).

ON	OFF	JUMPER
		USB_CP
		USB_DET
		SCL
		SDA
		ANALOG
		INT
		TEMP
		SPK
		LED
		LIGHT
		LCD
		CAN_T

**Figura 34:** *Impostazioni dei Jumper per il Progetto Hello\_World*

Non tutte le impostazioni sono in realtà importanti per ogni applicazione, nel nostro caso quello che realmente importa è che il display LCD sia disattivo e che i LED siano attivi.

Una volta che ci si è accertati che le impostazioni sono corrette è possibile procedere alla programmazione del PIC, premendo l'icona alla sinistra della Tool bar. Le altre icone sulla Tool bar permettono in particolare di svolgere le seguenti funzioni (partendo da sinistra):

- Programmare il PIC
- Leggere il PIC
- Leggere la EEPROM del PIC
- Verificare che il PIC contenga il progetto attuale
- Verificare che il PIC sia vuoto
- Rilasciare il PIC dallo stato di Reset
- Porre il PIC nello stato di Reset

Le ultime due funzioni sono particolarmente utili poiché permettono di rilasciare lo stato di Reset che si viene a creare quando si programma il PIC. Infatti dopo la programmazione il PIC non esegue il programma interno fino a quando non viene scollegato il programmatore o viene premuta la funzione di rilascio Reset. L'ultima funzione pone il PIC in stato di Reset; una nuova programmazione porrà in automatico il PIC nello stato di Reset.

Quando si rilascerà lo stato di Reset, se tutto è andato a buon fine si accenderà il LED1 ovvero 0.

---

## Capire come salutare il mondo

Dopo aver capito i vari passi da compiere per creare, compilare e installare il nostro progetto, è bene fare un passo in dietro per capire meglio il programma che abbiamo scritto. Tale spiegazione costruirà le basi sulle quali inizieremo poi la spiegazione più dettagliata del linguaggio C.

Normalmente il codice sorgente principale inizia con la direttiva `#include` nel nostro caso si ha:

```
#include <p18f4550.h>
```

Le direttive in C iniziano sempre con `#` e rappresentano delle indicazioni che il preprocessore (e non il compilatore) utilizzerà prima di avviare la compilazione del programma. Le direttive non appartengono al programma che il PIC fisicamente eseguirà; questo significa che una direttiva non è un'istruzione eseguibile dal PIC e non verrà dunque tradotta in codice eseguibile<sup>48</sup>. Per mezzo della direttiva `#include` si include il file `p18f4550.h`, questo file, come detto, contiene le informazioni del PIC che si sta utilizzando<sup>49</sup>. Ciò significa che a seconda del PIC che si utilizza il file da includere sarà diverso. Il file incluso e il file di linker devono essere tra loro coerenti al fine di evitare errori durante la compilazione. Come si vedrà in esempi successivi la direttiva `#include` viene anche utilizzata per includere file di libreria. Il codice successivo fa invece uso della direttiva `#pragma`:

```
#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF
```

A differenza della direttiva precedente che appartiene all'ANSI C, questa direttiva non è ANSI C. In questo caso la direttiva `#pragma` è utilizzata per modificare i registri di configurazione del PIC. Come visto durante la spiegazione dell'architettura PIC18, ogni PIC ha uno o più registri di configurazione (registri CONFIG) il cui scopo è settare l'hardware di base del PIC stesso in modo che possa funzionare correttamente ogni volta che viene alimentato il PIC. In particolare la direttiva `#pragma` può essere utilizzata per diversi scopi.

Per poter cambiare il valore dei registri di configurazione è necessario utilizzare l'opzione `config`, seguita dal nome dell'opzione da impostare. La particolare configurazione sarà impostabile a vari valori a seconda dell'hardware disponibile. Il nome delle opzioni e le loro possibili impostazioni sono mostrate per ogni PIC all'interno del file `hlpPIC18ConfigSet.chm` che è possibile trovare nella directory `doc` presente nella directory d'installazione del compilatore. Il nome delle opzioni non è sempre uguale per tutti i PIC, dunque è bene sempre fare riferimento al file indicato per maggior informazioni...soprattutto quando il compilatore segnala un errore del tipo opzione non valida. Le varie configurazioni è possibile impostarle, oltre che per mezzo del codice, ovvero utilizzando la direttiva `#pragma`, anche per mezzo del IDE stesso, selezionando dalla barra menù *Configure* → *Configuration bits*. Per mezzo di tale opzione verrà visualizzata la finestra d'impostazione visualizzata in Figura 35.

---

<sup>48</sup> Si noti che al termine della riga non è presente il punto e virgola, che invece verrà utilizzato al termine di ogni istruzione C.

<sup>49</sup> Tra le informazioni presenti nel file vi è il nome dei registri che in generale hanno lo stesso nome utilizzato anche sul datasheet del PIC che si sta utilizzando. L'utilizzo del nome dei registri permette di non considerare la locazione di memoria in cui sono fisicamente presenti permettendo dunque di raggiungere un livello di astrazione che semplifica il programma stesso.

Per abilitare la finestra è necessario rimuovere il check alla voce *Configuration Bits set in code*. Una volta abilitata è possibile selezionare ogni configurazione e sulla sua sinistra sarà possibile associare l'opzione voluta. Personalmente preferisco, forse per abitudine, l'utilizzo della direttiva `#pragma`. In questo modo le impostazioni appartengono al file e non al progetto, dunque se per qualunque ragione si vuole cambiare progetto non bisogna impostare nuovamente i bit di configurazione. Molti programmi venduti con i programmatori permettono anche di impostare i bit di configurazione prima di programmare il PIC, sovrascrivendo eventuali valori letti dal file .hex.

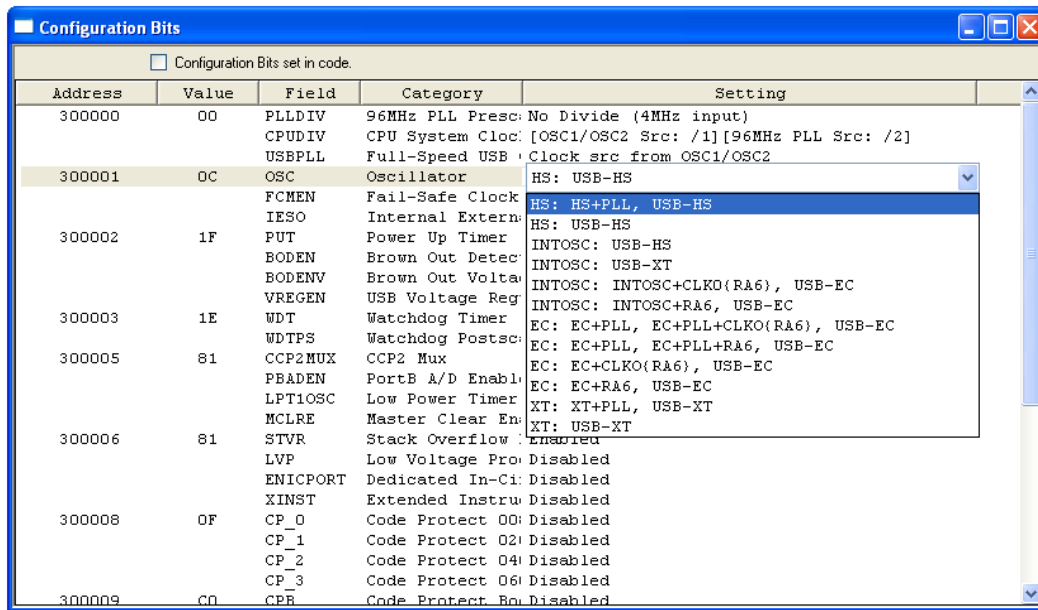


Figura 35: Tool Impostazione dei registri di Configurazione

Non sempre è necessario configurare tutti i bit, l'applicazione determinerà quello che è realmente necessario. Molti degli esempi proposti nel testo fanno uso di una configurazione piuttosto standard. Tra l'hardware sempre da impostare si ricorda l'oscillatore, in questo caso impostato ad HS, ovvero ad alta "velocità" questo poiché sul sistema Freedom II si è deciso di montare un quarzo da 20MHz. Qualora si voglia far uso del quarzo interno si dovrà selezionare l'opzione giusta.

Il WDT ovvero il Watchdog è disattivato avendo scritto OFF<sup>50</sup>. Anche la modalità LVP viene disattivata visto che non verrà utilizzata<sup>51</sup>.

In ultimo il bit PBADEN è posto ad OFF, in modo da utilizzare gli ingressi analogici presenti sulla PORTB come normali I/O digitali. Si fa presente che questo bit non è sempre presente su tutti PIC poiché non tutti i PIC18 hanno ingressi analogici anche sulla PORTB.

Quanto appena descritto è in realtà stato descritto parzialmente come commento subito dopo il codice. I commenti in C devono essere preceduti dal doppio slash `//`. I commenti così scritti devono però stare su di una sola riga. Se si volessero scrivere più righe di commento bisogna scrivere `//` all'inizio di ogni riga o scrivere `/*` alla prima riga e scrivere `*/` alla fine dell'ultima riga. Un possibile esempio di commento a riga multipla è il seguente:

```
// questo è un commento
```

<sup>50</sup> Per abilitare il WDT bisogna scrivere ON.

<sup>51</sup> La modalità LVP può risultare pratica qualora non sia disponibile la tensione normalmente utilizzata per programmare, normalmente più alta di quella di alimentazione. Nel nostro caso non è un problema poiché la tensione di programmazione è ottenuta direttamente dal programmatore.

---

```
// che non entrerebbe su una sola riga
```

o anche

```
/* questo è un commento  
che non entrerebbe su una sola riga */
```

o ancora

```
/* questo è un commento  
che non entrerebbe su una sola riga  
*/
```

o ancora

```
/*  
 * questo è un commento  
 * che non entrerebbe su una sola riga  
 */
```

Vediamo ora il programma vero e proprio. Ogni programma C è una collezione di funzioni<sup>52</sup> che possono o meno essere scritte sullo stesso file. Il numero di funzioni presente in ogni programma viene a dipendere dal programmatore e dal modo con cui ha organizzato la soluzione del suo problema. In ogni modo all'interno di ogni programma C deve essere sempre presente una funzione nominata `main`, una ed una sola funzione sarà chiamata `main`<sup>53</sup>.

La funzione `main` è quella che il compilatore andrà a cercare per prima e dalla quale organizzerà la compilazione<sup>54</sup> delle altre funzioni che saranno ragionevolmente collegate direttamente o indirettamente alla funzione `main`. Ogni funzione deve essere dichiarata nel modo seguente :

```
tipo_var_di_ritorno NomeFunzione (tipo_var_in_ingresso1)
```

La funzione `main` non ritorna nessun valore dunque viene scritto `void`, che sta ad indicare nessun valore. Inoltre la funzione `main` almeno per i PIC non accetta variabili in ingresso e dunque viene riscritto ancora `void`<sup>55</sup>. Come si vedrà in maggior dettaglio nel paragrafo dedicato alle funzioni, ogni funzione può ritornare al massimo un valore o puntatore ma può accettare in ingresso più variabili.

Ogni funzione, per sua natura, svolgerà qualche operazione che verrà poi tradotta dal compilatore in istruzioni eseguibili dal PIC. Queste istruzioni devono essere contenute all'interno di due parentesi graffe, una aperta e una chiusa, che stanno ad indicare l'inizio della funzione e la sua fine.

Per scrivere le parentesi graffe con tastiere italiane può esser un problema, questo problema non è sentito dagli americani (che guarda caso hanno inventato il C) poiché le parentesi graffe, come anche le quadre sono apparecchiate sulla tastiera. Un modo per scrivere le parentesi graffe è per mezzo dei codici ASCII 123 ( { ) e 125 ( } ). Per scrivere tali caratteri sulla tastiera bisogna tenere premuto il tasto ALT e digitare il codice 123 o 125, e poi rilasciare il tasto ALT. Un secondo modo, utilizzabile solo se si hanno le parentesi quadre

---

<sup>52</sup> Ogni funzione contiene un certo numero d'istruzioni per mezzo delle quali la funzione svolge il compito per cui è stata creata.

<sup>53</sup> Tutte le funzioni all'interno del nostro progetto devono essere individuate da un nome univoco.

<sup>54</sup> Quanto appena detto non deve essere preso per oro colato, ma è un modo valido per semplificare la trattazione dell'argomento.

<sup>55</sup> Si comprenderanno meglio le funzioni quando verranno utilizzate altre funzioni oltre al `main`.

---

sulla tastiera è quello di premere “freccia maiuscole + Alt Gr + parentesi quadra”.

Un programma che non fa nulla potrebbe essere scritto nel seguente modo:

```
void main (void) {  
  
    // Non faccio assolutamente nulla  
}
```

Personalmente metto le parentesi graffe come appena scritto, ma le si potrebbe anche scrivere in quest'altro modo.

```
void main (void)  
{  
  
    // Non faccio assolutamente nulla  
}
```

Per molto tempo ho personalmente usato questo secondo modo, ma dal momento che tutti i programmatori più noti fanno più uso della prima modalità, ho deciso di utilizzare la prima forma. L'aderire ad una versione o l'altra, non è di vitale importanza ma è importante che una volta scelta la preferita si mantenga lo stile scelto. Questo renderà il codice più leggibile, grazie alla sua uniformità. E' un qualcosa che apprezzerete quando dovrete scrivere del codice che va oltre qualche linea per salutare il mondo...e lo apprezzerete ancor più quando sarete costretti a rimettere mani sul codice a distanza di tempo.

Vediamo ora il programma che abbiamo scritto all'interno delle nostre parentesi graffe. Queste sono le prime righe di codice:

```
// Imposto PORTA tutti ingressi  
LATA = 0x00;  
TRISA = 0xFF;
```

Questo codice, da quanto spiegato nei Capitoli precedenti permette di inizializzare il latch di uscita della PORTA, ponendo i bit del registro LATA<sup>56</sup> a zero, ed imposta tutti i pin della PORTA come ingressi, infatti FF in esadecimale è 255 che in binario è 11111111, infatti ponendo a 1 il bit del registro TRIS si ha che il pin corrispondente sarà un input. E' possibile vedere che ogni istruzione termina con un punto e virgola<sup>57</sup>.

Il nome dei registri LATA e TRISA sono stati scritti in maiuscolo, questo non è stato fatto per metterli in evidenza ma poiché sono state dichiarate maiuscole. Infatti il compilatore C è case sensitive ovvero distingue tra maiuscole e minuscole. Dunque TRISA per il compilatore è diverso da TrisA.

Vediamo ora il valore che si è scritto nei registri LATA e TRISA. Il valore 0xFF rappresenta un valore alfanumerico espresso in esadecimale il suo formato è piuttosto standard. Ogni volta che si scrive un numero è importante scriverlo nella sintassi giusta, in modo da permettere al compilatore di riconoscere la base utilizzata (normalmente 2, 10 e 16)

---

<sup>56</sup> Si fa notare che il nome LATA non era stato definito da nessuna parte. Questo discende dal fatto che il nome è in realtà definito all'interno del file p18f4580.h

<sup>57</sup> Per chi ha esperienza di programmazione in Basic...e non, un errore tipico di sintassi è scordarsi il punto e virgola. Il compilatore in questo caso individua l'errore alla riga successiva a quella in cui manca effettivamente il punto e virgola. Da notare inoltre che il punto e virgola nella sintassi in assembler dichiara invece un commento.

---

e di conseguenza il valore rappresentato. Se si volesse per esempio impostare i bit RD0, RD1, RD2, RD3 come ingressi, mentre i bit RD4, RD5, RD6, RD7 come uscite, il valore da scrivere nel registro TRISD nei vari formati numerici sarebbe il seguente:

```
// Numero binario
TRISD = 0b00001111;

// Numero esadecimale
TRISD = 0x0F;

// Numero decimale
TRISD = 15;
```

Vi sarete forse chiesti per quale ragione la PORTA è stata impostata con tutti i pin come ingresso, anche se in realtà la PORTA non è affatto utilizzata nel nostro programma. La ragione è solo a scopo precauzionale, infatti, compatibilmente con l'hardware della scheda, è bene porre un pin non utilizzato come ingresso in modo da limitare problemi di cortocircuito accidentale.

Tutte le porte seguono le impostazioni appena descritte per la PORTA, tranne la PORTD, dal momento che è presente il LED con il quale si vuole salutare il mondo. I pin dei PIC possono pilotare senza problemi un LED standard fino a correnti massime di 20mA<sup>58</sup>.

Per ragioni pratiche, quando devo impostare un bit noto a 0 o 1 preferisco la rappresentazione binaria, visto che non è necessario sapere il valore rappresentato dal codice binario. In particolare per impostare RD0 come uscita si ha il seguente codice:

```
// Imposto PORTD tutti ingressi e RD0 come uscita
LATD = 0x00;
TRISD = 0b11111110;
```

Una volta inizializzati tutti i registri d'interesse viene acceso il LED ponendo ad 1 il bit del registro LATD, ovvero:

```
LATDbits.LATD0 = 1;
```

Un altro modo per accendere il LED, visto che il resto dei pin della PORTD è impostato come input, sarebbe potuto essere:

```
LATD = 0x01;
```

Ogni porta oltre ad avere il registro LATx possiede una variabile particolare (una struttura) nominata LATxbits che permette di accedere ogni singolo pin della porta stessa con il suo nome. Questa particolare variabile associata a LATx è presente anche per altri registri interni al PIC in cui si ha la necessità di leggere o scrivere singoli bit<sup>59</sup>. Una volta acceso il LED, è presente il seguente codice:

```
while (1) {
}
```

---

<sup>58</sup> Si ricorda che, anche se è possibile pilotare un led direttamente con il pin di uscita del PIC, ogni porta di uscita ha un limite massimo di corrente. Per maggiori informazioni sui limiti massimi di ogni porta si rimanda alle caratteristiche elettriche descritte nel datasheet del PIC utilizzato.

<sup>59</sup> Come detto il LED si sarebbe potuto accendere anche con il registro PORTD.



---

Attualmente la spiegazione del suo significato è un po' prematuro, ma vi basti sapere che l'istruzione rappresenta un ciclo infinito. Qualunque istruzione venisse messa all'interno del blocco delle parentesi graffe, verrebbe ripetuta all'infinito. Dal momento che non è presente nessuna istruzione vuol dire che il PIC è bloccato in un dolce far nulla. Qualora non venisse messo questo ciclo infinito, il programma terminerebbe e si riavvierebbe in automatico.

Istruzioni bloccanti di questo tipo sono anche necessarie quando si scrive un programma in C in ambiente Windows. Qualora non si bloccasse il programma per mezzo per esempio di una lettura da tastiera (tipo, premi un tasto per continuare) il terminale DOS verrebbe chiuso.

Per chi sa già programmare in C potrebbe obiettare che le parentesi graffe non servono, ed infatti è vero, si sarebbe potuto anche scrivere:

```
while (1);
```

In generale preferisco comunque scrivere le parentesi in modo da mettere in evidenza che il ciclo è un blocco vuoto. Questo risulterà molto utile per evitare brutte sorprese quando si aggiungerà nuovo codice. Di questo si parlerà comunque in seguito, ma rappresenta più che altro una regola di programmazione generale e non associata ai PIC.

Ultima nota va riguardo ai commenti. Frequentemente il commento viene visto sulla stessa riga del codice, tipo:

```
LATD = 0x00;          // Imposto PORTD tutti ingressi e RD0 come uscita
```

Nel testo il commento verrà invece scritto prima del codice, ovvero

```
// Imposto PORTD tutti ingressi e RD0 come uscita  
LATD = 0x00;
```

Questo risulta particolarmente pratico poiché permette al commento di seguire l'indentazione del codice, rendendo il tutto più leggibile. Anche in questo caso, come per le parentesi graffe il compilatore non avrà problemi a capire quando ha a che fare con del commento. Questo secondo metodo è consigliato poiché il più utilizzato a livello internazionale. All'interno del kernel Linux i programmatori preferiscono scrivere i commenti nel seguente modo:

```
/* Imposto PORTD tutti ingressi e RD0 come uscita */  
LATD = 0x00;
```

# Capitolo V



## Simuliamo i nostri programmi

In questo Capitolo si introdurranno le basi per poter utilizzare il simulatore presente all'interno dell'ambiente di sviluppo Microchip. Simulare può risultare particolarmente utile qualora si voglia vedere in dettaglio il comportamento del microcontrollore durante l'esecuzione del programma. Il suo valore didattico è inoltre particolarmente interessante, visto che permette di programmare ed eseguire programmi senza nemmeno avere una scheda di sviluppo.

### Perché e cosa simulare

Ogni volta che si realizza una nuova applicazione, indipendentemente da quanto tempo si sia passato ad analizzare il problema, si possono verificare difficoltà più o meno banali che sarà necessario risolvere. Avere una scheda di sviluppo sulla quale caricare il programma è certamente un aiuto, qualora sia possibile osservare fisicamente il problema. Qualora il problema dovesse essere non osservabile, come per esempio una comunicazione seriale che non funziona, risolvere il problema può risultare non banale. Gli approcci che vengono seguiti possono essere diversi, e l'esperienza del programmatore gioca sicuramente un ruolo importante alla risoluzione del problema.

La fase in cui il progettista passa a risolvere i problemi presenti sul proprio programma viene detta di Debug. Tra i tool per il Debug che la Microchip mette a disposizione vi è il simulatore MPLAB SIM che permette di emulare il microcontrollore a livello software, ovvero senza aver bisogno di schede di sviluppo dove caricare il programma. Una seconda modalità di Debug è offerta dai debugger, ovvero dispositivi connessi alla nostra scheda di sviluppo, che permettono di accedere ai registri interni del PIC durante l'esecuzione del programma. I programmatori Microchip della serie PICKIT 2, PICKIT 3, ICD possono tutti funzionare come debugger oltre che programmatori. Le performance del debugger sono in generale proporzionali al costo dello stesso. I limiti dei vari debugger sono generalmente associati al numero di Breakpoint che possono gestire e la velocità degli stessi.

Si fa notare che prima di simulare un programma, sia in modalità software che per mezzo del debugger, è necessario compilare il programma. Ogni programma può essere compilato sia in modalità Debug che modalità Release. La scelta dell'una o l'altra modalità può avvenire per mezzo della tool bar riportata in Figura 36.



**Figura 36:** Menù di scelta della modalità Debug o Release

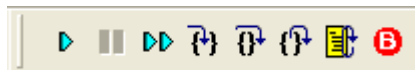
Le due modalità differiscono per l'abilitazione o meno di alcune ottimizzazioni. Durante la fase di Debug o simulazione è bene usare la modalità Debug e non quella di Release, visto

---

che il flusso del programma può risultare alterato a causa delle ottimizzazioni.

## Avvio di una sessione di simulazione

Per poter avviare una simulazione software non c'è bisogno della scheda di sviluppo dove poter caricare il programma. Dunque una volta creato il nostro progetto e compilato con successo, è possibile avviare una sessione di simulazione. Una sessione di simulazione viene avviata per mezzo del menù *Debugger* → *Select Tool* → *MPLAB SIM*. Qualora sia stato precedentemente impostato il programmatore esterno, si avrà un messaggio di avviso che dice che il programmatore e l'ambiente di simulazione non possono essere utilizzati in contemporanea. Premendo OK, il programmatore verrà disabilitato e verrà abilitato l'ambiente di simulazione MPLAB SIM. Il messaggio di avviso verrà nuovamente visualizzato quando si vorrà usare e riabilitare il programmatore. In quest'altro caso sarà l'ambiente di simulazione che verrà disabilitato. Una volta abilitato l'ambiente di simulazione, sulla tool bar compariranno i seguenti strumenti:



**Figura 37:** *Strumenti di simulazione*

Partendo dalla sinistra i pulsanti hanno il seguente significato:

- **Run**

Esegue il programma in maniera continua. Da un punto di vista visivo non è molto utile ma può certamente essere pratico quando si vuole simulare il controllo di determinate variabili e si stiano utilizzando i Breakpoint.

- **Pause**

Il tasto viene abilitato se il programma è stato messo in stato di run. In particolare tale tasto metterà in pausa il programma, potendo poi riprendere l'esecuzione anche *step by step*.

- **Animate**

Tale modalità è molto simile a quella *Run* ma la sua esecuzione avviene in tempi visibili ad occhio nudo, dando l'impressione appunto di un programma animato...

- **Step Into**

Rappresenta il comando *step by step*, ovvero le istruzioni vengono eseguite una alla volta. Un click equivarrà all'esecuzione di un'istruzione

- **Step Over**

Tale modalità di esecuzione permette di considerare la chiamata ad una funzione come se fosse una sola istruzione. Da un punto di vista funzionale, qualora non vi sono funzioni ad ogni click viene eseguita un'istruzione. Nel caso di chiamata ad una funzione, con un click, verranno eseguite tutte le istruzioni presenti nella funzione stessa. Se si è precedentemente usata la modalità *Step Into* e si è già all'interno della funzione, la modalità *Step Over* funziona come la *Step Into*, a meno di incontrare nuove funzioni.

---

- **Step Out**

Tale modalità di esecuzione permette, una volta entrati in una funzione per mezzo della modalità *Step Into*, di uscirne come se si fosse eseguita la modalità *Step Over*.

- **Reset**

Il tasto di Reset permette di inizializzare una nuova simulazione, riportando l'esecuzione del programma di nuovo all'inizio.

- **Breakpoint**

Permette d'inserire un Breakpoint, ovvero un punto di arresto. Questo permette di eseguire per esempio il programma in modalità *Run* (dunque veloce) fino al Breakpoint, normalmente un punto delicato dove è bene cominciare ad eseguire il programma passo passo. Nel caso di una simulazione software, diversamente dal caso in cui si faccia uso di un debugger, non vi sono limiti sul numero di Breakpoint che è possibile inserire.

Bene, a questo punto per iniziare la simulazione, ovvero l'esecuzione del programma possiamo iniziare a premere *Step Into*...

Alla prima pressione del tasto *Step Into*, come anche di un qualunque altro pulsante per avviare la simulazione, apparirà la finestra con il codice che verrà eseguito. E' possibile vedere che oltre al nostro codice in realtà viene eseguito altro codice del quale possiamo interamente ignorare le sue funzionalità...però diciamo qualche parola. Tale codice rappresenta il minimo indispensabile che è richiesto per inizializzare il nostro progetto, in particolare sarà questa parte di codice che chiamerà la nostra funzione `main`, ragion per cui la funzione `main` deve aver il nome `main`... Tale codice è presente all'interno del file `c018i.c` fornito con il compilatore dalla stessa Microchip.

```
// If user defined __init is not found, the one in clib.lib will be used
__init ();

// Call the user's main routine
main ();
```

Oltre alla chiamata della funzione `main`, che come visibile non ha parametri né in ingresso né in uscita, viene chiamata la funzione `__init ()`. Tale funzione è contenuta nel file `__init.c` ed è normalmente vuota. Qualora il programmatore volesse utilizzarla prima della chiamata alla funzione `main` potrebbe definire la sua funzione `__init ()`.

Continuando premere il tasto *Step Into*, sarà possibile vedere le varie istruzioni che verranno eseguite. Ad un tratto verrà eseguita la chiamata alla funzione `main`, a questo punto possiamo eseguire la simulazione del nostro programma.

Durante la fase di simulazione vengono offerti diversi strumenti utili per mezzo dei quali è possibile vedere il valore delle variabili e registri all'interno del PIC, permettendo dunque di individuare eventuali anomalie. La descrizione su come utilizzare ed impostare tali funzioni verrà descritta a breve dal momento che è praticamente comune anche alla modalità di Debug.

## Avvio di una sessione di Debug

Per poter avviare una sessione di Debug, diversamente dalla simulazione software, sarà necessario possedere il sistema di sviluppo programmato o meno (visto che si potrà programmare durante la fase di Debug), ed di un debugger compatibile con l'ambiente di

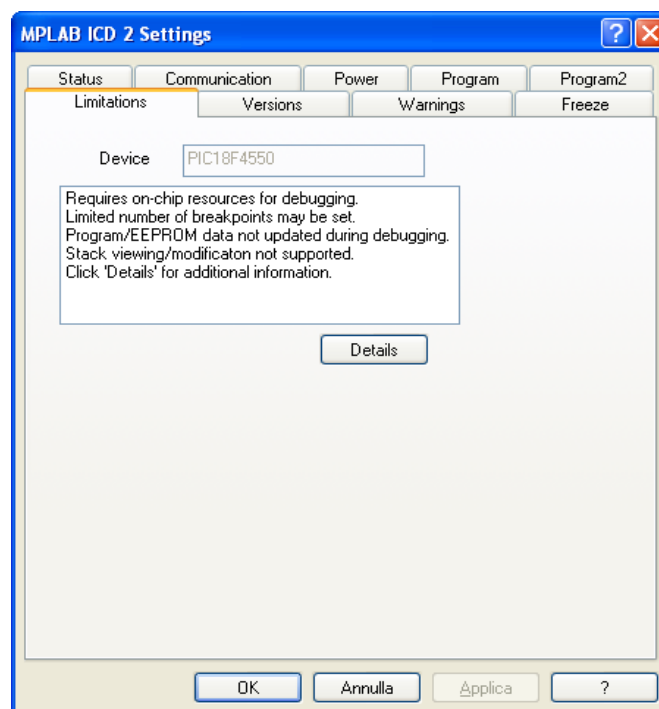
sviluppo Microchip. A questo punto se siete stati ostinati a non prendere strumenti compatibili Microchip non avrete modo di testare la modalità di Debug. Nella discussione che segue si prenderà come esempio il debugger ICD 2 ma altri debugger lavoreranno in maniera analoga.

Una volta che l'hardware è connesso è possibile abilitare la sessione di Debug per mezzo del menù *Debugger* → *Select Tool* → *ICD 2*. Qualora sia stato precedentemente impostato il programmatore esterno, si avrà un messaggio di avviso che dice che il programmatore e l'ambiente di simulazione non possono essere utilizzati in contemporanea. Premendo OK, il programmatore verrà disabilitato e verrà abilitato l'ambiente di Debug. Il messaggio di avviso verrà nuovamente visualizzato quando si vorrà usare e riabilitare il programmatore. In quest'altro caso sarà l'ambiente di simulazione che verrà disabilitato. Una volta abilitato l'ambiente di simulazione, sulla tool bar compariranno i seguenti strumenti<sup>60</sup>:



**Figura 38:** *Strumenti di Debug*

Questa nuova barra degli strumenti, oltre alla barra precedentemente comparsa per la simulazione, possiede gli strumenti base precedentemente descritti per la programmazione. Inoltre sulla destra è possibile vedere che sul debugger utilizzato è possibile far uso di 3 Breakpoint, di cui attualmente 0 sono usati. Come detto ogni debugger possiede delle limitazioni, in particolare le limitazioni che si hanno durante una fase di Debug possono essere lette dal menù *Debugger* → *Settings*, come riportato in Figura 39. Per mezzo di questa finestra di dialogo è anche possibile ottenere altre informazioni riguardo al debugger utilizzato. Una opzione utile è quella relativa alle tensioni disponibili sulla scheda di sviluppo visualizzabili dal Tab Power.



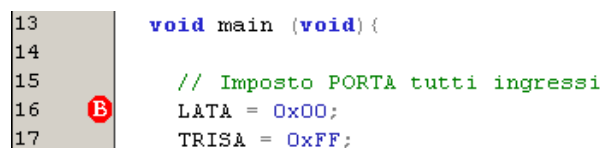
**Figura 39:** *Limitazioni della sessione di Debug*

<sup>60</sup> All'avvio la barra degli strumenti è disabilitata; sarà necessario premere il tasto della barra *Reset and Connect to ICD*, per poter attivare la barra.

Una volta caricato il programma di cui si vuole fare il Debug è possibile iniziare la simulazione premendo il tasto *Step Into*. Come nel caso della simulazione software il programma inizierà dalle inizializzazioni dietro le quinte. In particolare ad ogni esecuzione di un'istruzione si vedrà il debugger entrare in stato di *Busy* (occupato) per mezzo dell'accensione del LED arancione posto sul contenitore.

## Utilizzo dei Breakpoint

I Breakpoint rappresentano dei punti di arresto per il programma. La loro gestione è identica sia per la simulazione software che la fase di Debug. In quest'ultima però sono presenti delle limitazioni che vengono a dipendere dal debugger utilizzato. Per abilitare un punto di blocco, basta fare un doppio click sul numero della linea di codice<sup>61</sup>. Quando si sarà inserito il punto di blocco al fianco della linea di codice si avrà una B rossa come riportato in Figura 40.



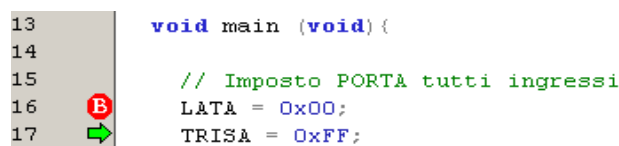
```
13 void main (void) {
14
15     // Imposto PORTA tutti ingressi
16     LATA = 0x00;
17     TRISA = 0xFF;
```

A screenshot of a code editor with a light gray background. On the left, line numbers 13 through 17 are listed. To the right of the code, a red circle with a white 'B' is positioned next to line 16, indicating a breakpoint. The code itself is: `void main (void) {` on line 13, an empty line on 14, a comment `// Imposto PORTA tutti ingressi` on 15, `LATA = 0x00;` on 16, and `TRISA = 0xFF;` on 17.

Figura 40: Posizionamento di un punto di blocco

Se il Breakpoint verrà messo all'inizio della funzione main o in una linea senza codice, il Breakpoint verrà ignorato.

A questo punto se si attiva la modalità di esecuzione *Run*, il programma verrà eseguito fino al punto del nostro punto di blocco, come riportato in Figura 41. In particolare la freccia verde mostra che l'istruzione alla riga 16 è stata eseguita e la prossima sarà la riga 17.



```
13 void main (void) {
14
15     // Imposto PORTA tutti ingressi
16     LATA = 0x00;
17     TRISA = 0xFF;
```

A screenshot of the same code editor as in Figure 40. A red circle with a white 'B' is next to line 16. A green arrow points to line 17, indicating that the program has executed up to the breakpoint on line 16 and is now paused before line 17.

Figura 41: Esecuzione del programma fino al punto di blocco

Per mezzo del Breakpoint abbiamo in questo esempio saltato la noiosa parte di premere molte volte *Step Into*, prima di eseguire il nostro programma. Lo stesso risultato si può ottenere posizionando il cursore del mouse sulla riga di codice dove si vuole iniziare *Step Into* manualmente e premere il tasto destro. Dal pop-up menù selezionare poi la voce *Run to Cursor*, come visibile in Figura 42.

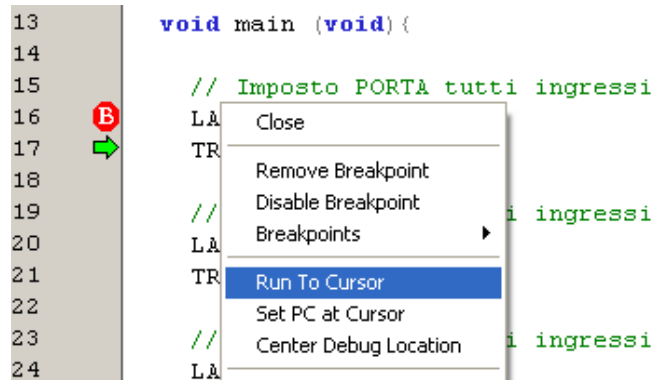
Ritornando ai Breakpoint, come detto, è possibile nel caso del nostro debugger far uso di soli 3 Breakpoint. In realtà è comunque possibile inserirne altri e lasciarli disabilitati<sup>62</sup>. Per fare questo bisogna aprire la finestra di dialogo per la gestione dei Breakpoint, premendo la B rossa sulla Tool bar. La finestra di dialogo che si viene ad aprire è riportata in Figura 43. Dalla finestra di gestione Breakpoint è possibile inserirne di nuovi, eliminarli e disabilitarli<sup>63</sup>.

<sup>61</sup> Il doppio click di default abilita il Breakpoint, ciononostante è possibile cambiare le opzioni dell'editor al fine da far funzionare il doppio click come per un normale editor, ovvero per selezionare la parola.

<sup>62</sup> In particolare inserendo più Breakpoint di quelli gestibili, causa l'inserimento degli altri in modalità disabilitata.

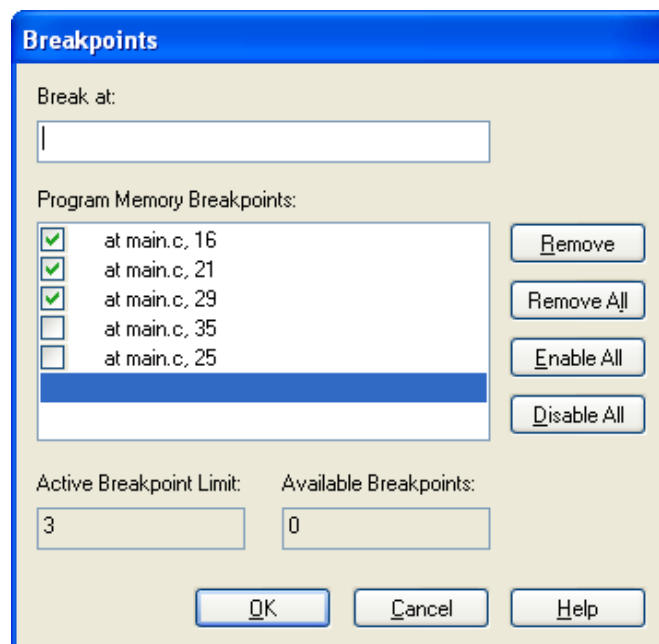
<sup>63</sup> Un Breakpoint disabilitato è rappresentato da un ottagono vuoto con bordo rosso.

E' possibile notare che nel nostro caso solo 3 Breakpoint possono essere abilitati allo stesso tempo. Da quanto spiegato è possibile capire che la funzione dei Breakpoint è di porre un punto di blocco alla normale esecuzione del programma in modalità *Run* o *Animate*. Se infatti si sta eseguendo il programma in modalità *Step Into* il Breakpoint viene ignorato.



**Figura 42:** Utilizzo della modalità *Run to Cursor*

Una volta che il programma è stato arrestato su un punto di blocco, è possibile riprendere la sua esecuzione in qualunque modalità, sia essa automatica o *Step by Step*.

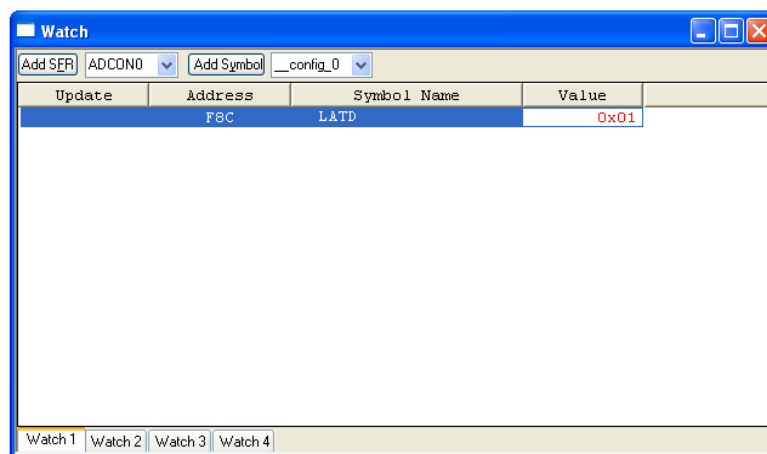


**Figura 43:** Finestra di gestione Breakpoint

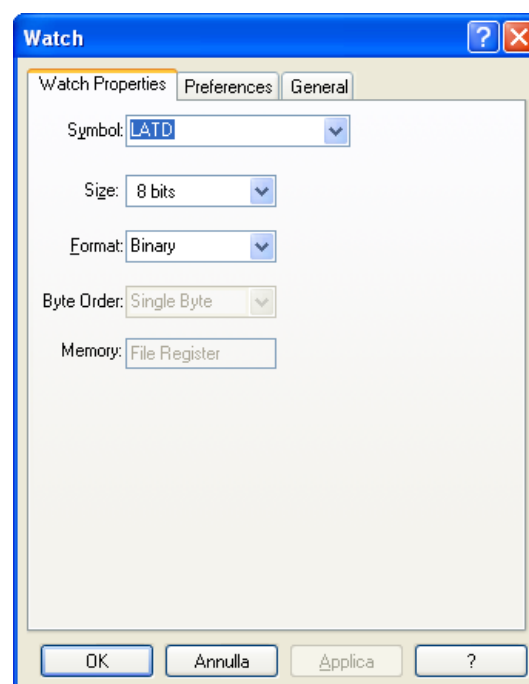
## Controllo delle variabili e registri

Poter eseguire il programma passo passo e bloccarlo a nostro piacimento è sicuramente utile, ma se non si possono vedere i contenuti delle variabili e registri interni al PIC quanto fin ora spiegato non avrebbe alcuna utilità. Una delle funzioni principali della sessione di simulazione software e di Debug, è proprio quella di poter scrutare i contenuti dei registri e

delle variabili definite nel nostro programma. In questo modo è possibile capire per quale ragione un LED non si accende o per quale ragione una periferica non sta funzionando correttamente. Uno dei modi più pratici per la visualizzazione delle variabili è per mezzo della finestra di *Watch*, visualizzabile dal menù *View* → *Watch*, un esempio di finestra *Watch* in cui viene visualizzato il registro LATD è riportato in Figura 44. All'interno di questa finestra è possibile aggiungere per mezzo del tasto *Add SFR*, un qualunque registro SFR, come ad esempio LATD. Per mezzo del tasto *Add Symbol* è invece possibile inserire una qualunque variabile dichiarata nel programma. Il contenuto della variabile o registro viene visualizzato all'interno della colonna *Value*. Il suo formato può essere cambiato a seconda delle esigenze semplicemente selezionando la variabile o registro e premendo il tasto destro del mouse. Dal pop up menù selezionare poi *Properties*. La finestra di Dialogo delle proprietà è quella riportata in Figura 45.



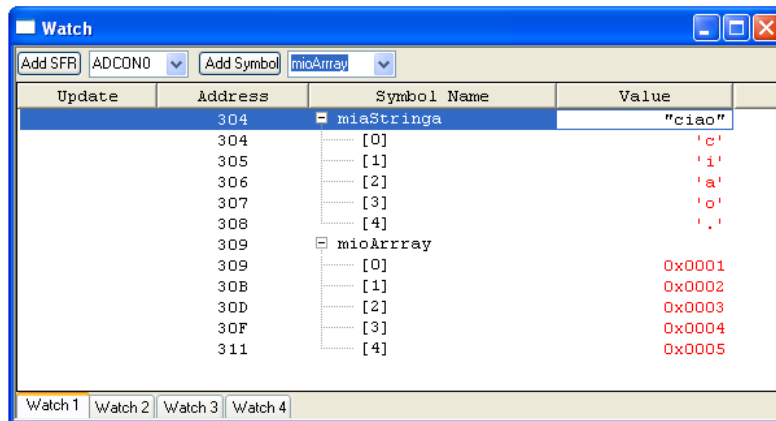
**Figura 44:** Finestra di visualizzazione delle variabili



**Figura 45:** Finestra d'impostazione del formato delle variabili



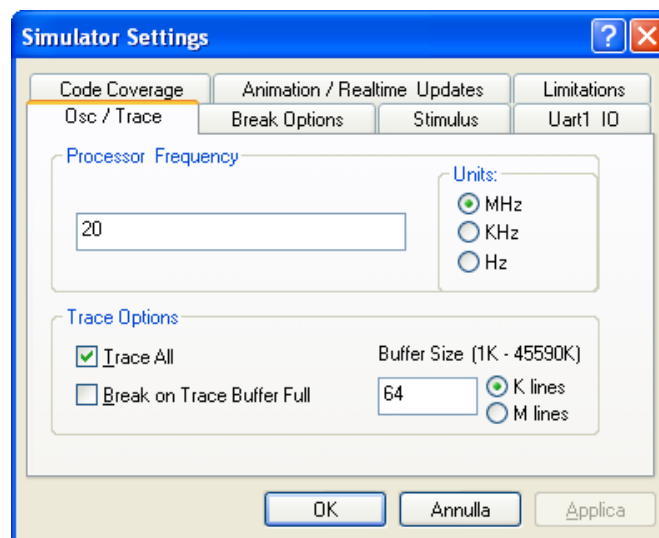
Oltre alla finestra Watch è possibile visualizzare altre tipologie di finestre con diversi contenuti, tutte richiamabili dal menù *View*. Il loro utilizzo è piuttosto semplice dunque lascio a voi il loro studio ed utilizzo. Nel caso si visualizzino strutture dati più complesse, quale per esempio gli Array, la visualizzazione delle variabili risulta molto pratica, come visibile in Figura 46.



**Figura 46:** Finestra di visualizzazione in caso di Array

## Analisi temporale

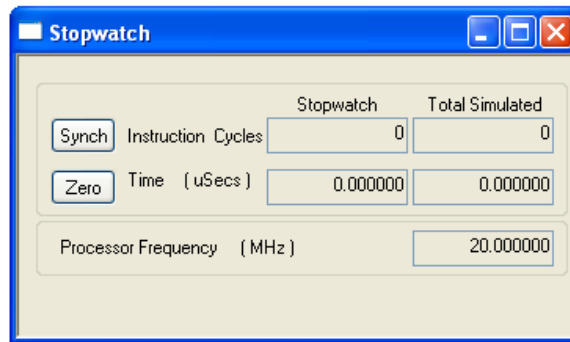
Quando si scrivono programmi in cui il tempo di esecuzione di una determinata procedura può influenzare il risultato o la funzionalità del programma stesso, risulta particolarmente utile la possibilità di controllare il tempo di esecuzione di una determinata funzione o parte di programma. Si capisce innanzitutto che il tempo di esecuzione viene a dipendere dalla frequenza del clock, dunque dal periodo del ciclo istruzioni che equivale come detto a 4 cicli di clock. Il clock principale può essere generato sia da un quarzo esterno che dall'oscillatore interno. Indipendentemente dalla sorgente avrà un suo valore. Tale valore deve essere in un certo qual modo reso disponibile al simulatore. Per scrivere tale informazione si avvia il simulatore Software MPLAB SIM<sup>64</sup>, e si vada al menù *Debugger* → *Settings*. Verrà visualizzata la finestra di dialogo di Figura 47.



**Figura 47:** Finestra di dialogo per le impostazioni del Simulatore MPLAB SIM

Al Tab *Osc/Trace* viene visualizzata la frequenza alla quale verrà fatto operare il PIC. Tale valore nel caso della scheda Freedom II, e nel caso in cui si faccia uso del quarzo esterno, è pari a 20MHz. Per valori differenti non si deve far altro che scrivere il valore nella casella di testo. Una volta impostato il valore del quarzo è possibile per mezzo dello strumento eseguibile dal menù *Debugger* → *StopWatch* riportato in Figura 48, controllare i tempi di esecuzione di un programma o parte di esso.

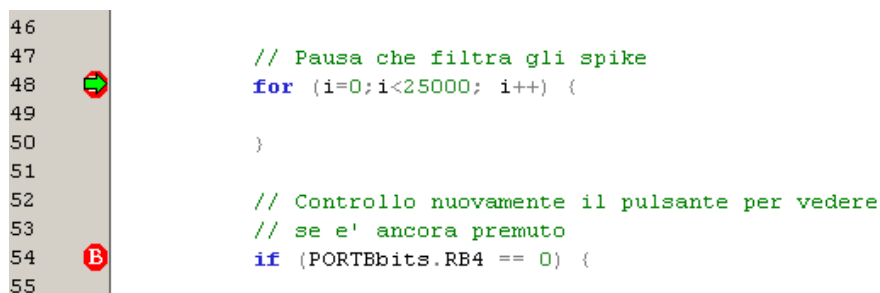
<sup>64</sup> Quanto verrà spiegato non è infatti disponibile per il debugger.



**Figura 48:** *Strumento StopWatch*

Questo semplice ma funzionale strumento, possiede due soli pulsanti. Il primo pulsante è nominato *Zero* e permette di azzerare il conteggio mentre il secondo pulsante *Synch*, permette di sincronizzare il conteggio con il tempo totale di esecuzione del programma.

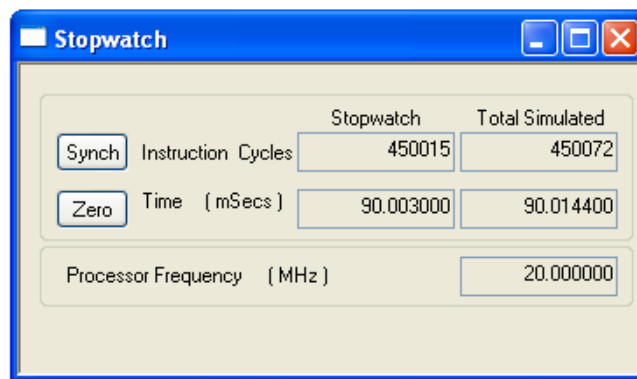
Se per esempio si volesse conoscere la pausa introdotta per mezzo di un ciclo for con conteggio a vuoto (si parlerà in maggior dettaglio del ciclo for, nei prossimi Capitoli) è possibile mettere due Breakpoint uno all'inizio del conteggio e uno alla prima istruzione successiva al conteggio, come riportato in Figura 49.



**Figura 49:** *Posizionamento dei Breakpoint*

Una volta effettuato il *Reset* del simulatore sarà possibile eseguire il comando *Run*. Il puntatore verrà a fermarsi proprio sulla posizione voluta. A questo punto il nostro strumento *StopWatch* avrà già contato dei tempi di esecuzione, che a noi non interessano, per questo motivo lo si azzerà per mezzo del tasto *Zero*. A questo punto è possibile rieseguire il comando *Run*, che nuovamente si arresterà sul nuovo Breakpoint.

I valori ora riportati sullo strumento *StopWatch*, riportato in Figura 50, hanno adesso un significato di un certo interesse. Il valore 450015 rappresenta il numero di istruzioni eseguite per implementare il ciclo for (più 1 perché è stata eseguita l'istruzione alla riga 54). Si può notare subito che ripetendo 25000 volte il ciclo si eseguendo in realtà 450015 istruzioni! Il tempo di esecuzione è di circa 90ms, ovvero la pausa introdotta è circa un decimo di secondo. Con due passi abbiamo scoperto il ritardo introdotto. Se lo si volesse non di 90ms ma di 100ms, ovvero di un decimo di secondo, si potrebbe, cambiando la variabile d'incremento da 25000 a 27777 ottenere un ritardo di 100.000200. Cercare di ottenere tempi troppo precisi può essere superfluo infatti per ottenere una base tempi precisa bisognerebbe lavorare molto sulla precisione del clock stesso.



**Figura 50:** *Strumento StopWatch*

Prima di modificare la circuiteria di clock, si potrebbe ottimizzare il codice facendo uso di una routine assembler, in modo da gestire meglio le singole istruzioni. Gli oscillatori di precisione che è possibile trovare in commercio possono raggiungere le diverse centinaia di euro. Se proprio si volesse spendere di più si possono tranquillamente trovare oscillatori da qualche migliaia di euro. Come sempre uno dei limiti principali è il basso costo mentre per chi volesse spendere molto non c'è mai un limite...visto che non ho considerato i clock atomici!

# Capitolo VI

## Impariamo a programmare in C

Forti dell'esperienza e delle conoscenze dei Capitoli precedenti verrà ora introdotta con maggior dettaglio la sintassi e le parole chiave del linguaggio C. Questo verrà spiegato partendo da zero, non facendo affidamento a nessun background che non sia basato sui Capitoli precedenti. In ogni modo qualora il lettore abbia familiarità con il C o altri linguaggi di programmazione, si troverà sicuramente avvantaggiato e troverà di più facile comprensione quanto verrà spiegato.

### La sintassi in breve

Nel nostro primo progetto abbiamo già visto alcuni aspetti legati alla sintassi del C, ovvero a quell'insieme di regole che permettono di strutturare il programma in maniera tale che il compilatore possa propriamente interpretarle. Il C possiede in particolare delle parole chiave ovvero keyword che è in grado d'interpretare. Ogni programma rappresenta una combinazione più o meno complessa di tali parole chiave più delle istruzioni. Il C18 è conforme allo standard ANSI C dunque possiede le seguenti parole chiave:

Parole Chiave ANSI C	
auto	int
break	long
case	register
char	return
const	short
continue	signed
default	sizeof
do	static
double	struct
else	switch
enum	typedef
extern	union
float	unsigned
for	void
goto	volatile
if	while

**Tabella 4:** Parole chiave dell'ANSI C

---

Ogni parola chiave, qualora si stia facendo uso dell'IDE MPLAB viene automaticamente riconosciuta e messa in grassetto<sup>65</sup>. Nel seguente testo verrà illustrato l'utilizzo delle parole chiave più utilizzate. Essendo il C18 conforme allo standard ANSI C un qualunque testo di C sarà sufficiente per comprendere le parole chiave non trattate.

Come detto il programma oltre che essere composto da parole chiave è composto da istruzioni ovvero operazioni da compiere. Ogni istruzione deve essere terminata per mezzo di un punto e virgola.

```
a = b + c;
```

Su ogni linea è possibile scrivere anche più istruzioni, purché siano tutte sperate da un punto e virgola:

```
a = b + c; d = e - a; f = d;
```

Si capisce che in questo modo...non si capisce nulla, dunque è bene scrivere il codice in maniera da avere un'istruzione per linea:

```
a = b + c;
d = e - a;
f = d;
```

In questo modo si aumenta molto la leggibilità del codice stesso. Negli esempi che seguiranno verrà messo più volte in evidenza che la leggibilità del codice è molto importante.

Alla regola del punto e virgola fanno eccezione le direttive, dal momento che non sono delle istruzioni che il compilatore tradurrà in codice macchina.

```
#include <p18f4550.h>
```

Alcune parole chiave fanno utilizzo di blocchi d'istruzione ovvero insieme d'istruzioni. Un blocco è identificato da parentesi graffe:

```
if (a < b) {
    d = e - a;
    f = d;
}
```

Come si vedrà in seguito l'insieme d'istruzioni nelle parentesi graffe viene eseguito solo se a è minore di b.

Le varie linee di codice ammettono commenti che possono essere introdotti per mezzo di varie sintassi. Il loro utilizzo è stato già introdotto e verrà continuamente mostrato nel seguito del testo. Anche in questo caso l'IDE MPLAB riconosce in automatico i commenti e li colora normalmente di verde.

Una caratteristica particolare del C che lo differenzia da altri linguaggi di programmazione,

---

<sup>65</sup> Per mezzo delle proprietà dell'editor è possibile cambiare il colore ed altre proprietà associate alle varie tipologie di testo e caratteri che l'IDE riconosce automaticamente.

---

è il fatto che è un linguaggio *case sensitive*, ovvero sensibile alle maiuscole e minuscole. Questo significa che le parole chiave appena introdotte, se scritte in maniera differente non saranno riconosciute come tali. Quanto detto va inoltre generalizzato anche al nome di variabili; per esempio se consideriamo:

```
int temperatura = 0;
```

risulta differente dalla variabile:

```
int Temperatura = 0;
```

In generale anche se le due variabili risultano differenti si sconsiglia di utilizzare questa tecnica per differenziare delle variabili all'interno dello stesso spazio di utilizzo (ovvero scope, come verrà spiegato nei paragrafi successivi).

## Tipi di variabili

Ogni volta che si scrive un programma, salvo il caso precedente, è necessario svolgere delle operazioni che richiedono dei registri per memorizzare il risultato o le variabili per l'operazione stessa. I registri, ovvero variabili, non sono altro che locazioni di memoria RAM interna al microcontrollore che come abbiamo visto viene ottenuta per mezzo di RAM statica con locazioni di memoria ad 8 bit, ovvero un byte. Il numero di registri necessari per rappresentare un tipo di variabile varia a seconda del tipo di variabile.

Le tipologie di variabili disponibili in C che è possibile utilizzare nella programmazione dei PIC sono riportate in Tabella 5 e Tabella 6.

Tipo	Dimensione	Minimo	Massimo
char	8 bits	-128	+127
signed char	8 bits	-128	+127
unsigned char	8 bits	0	255
int	16 bits	-32.768	+32.767
unsigned int	16 bits	0	65.535
short	16 bits	-32.768	+32.767
unsigned short	16 bits	0	65.535
short long	24 bits	-8.388.608	+8.388.607
unsigned short long	24 bits	0	16.777.215
long	32 bits	-2.147.483.648	2147483647
unsigned long	32 bits	0	4.294.967.295

**Tabella 5:** *Tipi di variabili disponibili*

Come è possibile osservare a seconda del tipo di variabile, la dimensione, ovvero il numero di byte necessari per contenere la variabile, è diverso. Inoltre al variare del tipo di variabile, anche il valore minimo e massimo che è possibile contenere nella variabile, ovvero rappresentare, cambia. Si capisce dunque che a seconda del tipo di dato che bisogna memorizzare e del valore numerico minimo o massimo, bisogna scegliere un tipo di variabile piuttosto che un'altra. Una scelta attenta permetterà di risparmiare locazioni di memoria RAM e anche il tempo necessario per lo svolgimento delle operazioni. Infatti fare una somma tra due interi contenuti in una variabile char sarà più veloce che fare la somma tra due interi di tipo int. Quanto detto può essere visto con questo semplice esempio in cui si dichiarano delle variabili char e si fa la somma tra loro. Lo stesso esempio è poi ripetuto per mezzo di variabili di tipo int. Nell'esempio è possibile osservare che per dichiarare una variabile di un certo tipo è necessario precedere il nome della variabile con il tipo.

```
// Dichiarazione delle variabili
char charA;
char charB;
char charC;

int intA;
int intB;
int intC;

// Esempio inizializzazione e somma tra char
```



```

charA = 5;
charB = 8;

charC = charA + charB;

// Esempio inizializzazione e somma tra int
intA = 5;
intB = 8;

intC = intA + intB;

```

Per verificare se quanto abbiamo detto è vero bisognerà vedere il codice assembler ottenuto dopo la compilazione del programma. Tale tecnica è in generale utile durante la fase di ottimizzazione del codice, per verificare se effettivamente valga o meno la pena modificare delle sue parti. Si fa notare inoltre che questo approccio è utile qualunque sia il linguaggio di programmazione utilizzato. In ogni modo è bene far notare che non bisogna ottimizzare il codice sin dall'inizio<sup>66</sup>, poiché il codice ottimizzato frutto di idee brillanti, può in generale essere di più difficile lettura...soprattutto a distanza di mesi<sup>67</sup>.

Per verificare il codice assembler associato ad una compilazione è possibile aprire la finestra *Disassembly Listing* dal menù *View*. In tale finestra sarà mostrato il nostro codice e il risultato della compilazione. In particolare è possibile leggere gli stessi commenti scritti nel codice sorgente e le stesse istruzioni in C con relativa decodifica. Dal codice sotto riportato è possibile subito vedere che la parte associata all'assegnazione dei valori nelle variabili *char* (ovvero la loro inizializzazione) risulta molto semplice, e come ci si poteva aspettare coinvolge due semplici movimenti con il registro W (accumulatore). Uno per caricare la costante e l'altro per caricare tale valore nel registro voluto. La somma tra i due registri risulta un po' più laboriosa dell'inizializzazione, ma pur sempre piuttosto semplice.

```

43:                                     // Esempio inizializzazione e somma tra char
44:                                     charA = 5;
    00C6      0E05      MOVLW 0x5
    00C8      6EDF      MOVWF 0xfdf, ACCESS
45:                                     charB = 8;
    00CA      52DE      MOVF 0xfde, F, ACCESS
    00CC      0E08      MOVLW 0x8
    00CE      6EDD      MOVWF 0xfdd, ACCESS
46:
47:                                     charC = charA + charB;
    00D0      0E01      MOVLW 0x1
    00D2      50DB      MOVF 0xfdb, W, ACCESS
    00D4      24DF      ADDWF 0xfdf, W, ACCESS
    00D6      6EE7      MOVWF 0xfe7, ACCESS
    00D8      0E02      MOVLW 0x2
    00DA      CFE7      MOVFF 0xfe7, 0xfdb
    00DC      FFDB      NOP

```

Nel caso di somma tra due interi si vede subito che l'aver utilizzato un tipo di variabile *int* comporta una maggiore complessità del codice ad esso associato. Naturalmente a noi interesserà poco di capire riga per riga, poiché quello che facciamo è semplicemente dichiarare delle variabili e fare la loro somma. In ogni modo è bene tenere a mente che un utilizzo corretto delle variabili può portare ad un codice migliore, permettendo non solo di

<sup>66</sup> Anche se in generale non si consiglia di ottimizzare il codice sin dall'inizio, in applicazioni embedded è sempre bene tenere a mente che le risorse sono limitate!

<sup>67</sup> Nei casi in cui si faccia uso di soluzioni brillanti è sempre bene documentare il tutto con buoni commenti. L'esser brillanti è ristretto a frangenti temporali che è difficile ripetere...gran parte delle volte abbiamo bisogno di una bella spiegazione per raggiungere quello che un momento di lucidità ha richiesto pochi secondi!

risparmiare memoria di programma (poiché si utilizzano meno istruzioni), ma anche tempi di esecuzioni minori.

```
49:                                     // Esempio inizializzazione e somma tra int
50:                                     intA = 5;
   00DE    0E05    MOVLW 0x5
   00E0    6EF3    MOVWF 0xff3, ACCESS
   00E2    0E03    MOVLW 0x3
   00E4    CFF3    MOVFF 0xff3, 0xfdb
   00E6    FFDB    NOP
   00E8    0E04    MOVLW 0x4
   00EA    6ADB    CLRF 0xfdb, ACCESS
51:                                     intB = 8;
   00EC    0E08    MOVLW 0x8
   00EE    6EF3    MOVWF 0xff3, ACCESS
   00F0    0E05    MOVLW 0x5
   00F2    CFF3    MOVFF 0xff3, 0xfdb
   00F4    FFDB    NOP
   00F6    0E06    MOVLW 0x6
   00F8    6ADB    CLRF 0xfdb, ACCESS
52:
53:                                     intC = intA + intB;
   00FA    0E05    MOVLW 0x5
   00FC    CFDB    MOVFF 0xfdb, 0x2
   00FE    F002    NOP
   0100    0E06    MOVLW 0x6
   0102    CFDB    MOVFF 0xfdb, 0x3
   0104    F003    NOP
   0106    0E03    MOVLW 0x3
   0108    50DB    MOVF 0xfdb, W, ACCESS
   010A    2402    ADDWF 0x2, W, ACCESS
   010C    6E00    MOVWF 0, ACCESS
   010E    0E04    MOVLW 0x4
   0110    50DB    MOVF 0xfdb, W, ACCESS
   0112    2003    ADDWFC 0x3, W, ACCESS
   0114    6E01    MOVWF 0x1, ACCESS
   0116    0E07    MOVLW 0x7
   0118    C000    MOVFF 0, 0xfdb
   011A    FFDB    NOP
   011C    0E08    MOVLW 0x8
   011E    C001    MOVFF 0x1, 0xfdb
   0120    FFDB    NOP
```

La variabile di tipo `char` anche se formalmente è pensata per contenere il valore numerico di un carattere ASCII può risultare utile in molti altri casi. Basti infatti pensare che le porte di uscita del PIC sono a 8 bit, dunque ogni volta che bisogna memorizzare qualche tipo di dato che deve essere poi posto in uscita, la variabile `char` è sicuramente un buon candidato.

Per mezzo del C è anche possibile effettuare divisioni; per poter memorizzare il risultato è in generale richiesto un numero decimale. Per questa esigenza sono presenti altri due tipi di variabili dette floating point come riportato in Tabella 6<sup>68</sup>.

Le variabili floating point a differenza delle variabili intere sono caratterizzate da una mantissa e da un esponente e dal fatto che non tutti i valori compresi nell'intervallo minimo e massimo sono in realtà possibili (si ha una granularità). Questo significa che un risultato

<sup>68</sup> E' possibile osservare che le due variabili, come anche per alcune variabili intere, non sono differenti. Eventuali differenze sono generalmente legate al compilatore utilizzato.

floating point è in generale il valore più prossimo al risultato reale<sup>69</sup>.

Tipo	Dimensione	Exp. min	Exp. max.	Min. normalizzato	Max. normalizzato
float	32 bits	-126	+128	$2^{-126} \approx 1.17549435e^{-38}$	$2^{128} * (2-2^{-15}) \approx 6.80564693e^{+38}$
double	64 bits	-126	+128	$2^{-126} \approx 1.17549435e^{-38}$	$2^{128} * (2-2^{-15}) \approx 6.80564693e^{+38}$

**Tabella 6:** Tipi di variabili disponibili II

Ogni volta che si dichiara una variabile è molto importante inizializzarla ovvero dargli un valore iniziale, che generalmente è il valore nullo. Questo può essere fatto sia in fase di dichiarazione della variabile che successivamente. L'esempio che segue mostra come inizializzare nei due modi le variabili *i* e *x*.

```
void main (void) {  
  
    // Dichiarazione della variabile intera i  
    int i;  
  
    // Dichiarazione ed inizializzazione della variabile intera x  
    int x = 0;  
  
    // Inizializzazione della variabile intera i  
    i = 0;  
  
}
```

Le due modalità d'inizializzazione delle variabili sono del tutto equivalenti ai fini pratici. E' necessario osservare che la dichiarazione delle variabili deve essere fatta all'inizio, ogni funzione sia essa la funzione `main ( )` o un'altra, deve dichiarare le sue variabili prima d'iniziare a svolgere una qualunque operazione<sup>70</sup>.

Le variabili fin ora introdotte sono sufficienti per organizzare ogni tipo di programma, ma spesso ci sono grandezze per le quali è utile avere più variabili dello stesso tipo. Si pensi ad esempio ad un programma che debba memorizzare la temperatura ogni ora. Piuttosto che dichiarare 24 variabili dello stesso tipo risulta utile dichiarare una sola variabile che permetta di contenerle tutte. Questi tipi di variabili si chiamano Array o vettori, e possono essere anche multidimensionali<sup>71</sup>. Nel caso preso in esame si ha a che fare con un Array monodimensionale poiché un indice è sufficiente ad individuare tutti i suoi elementi. In C18 per dichiarare un Array si procede come per il C:

```
void main (void) {  
  
    // Array di caratteri con 10 elementi  
    char mioArray[10];  
  
}
```

<sup>69</sup> Il fatto che un numero floating point non è sempre un valore esatto del risultato è spesso sottovalutato. In algoritmi in cui viene calcolato in maniera iterativa un risultato si può avere la divergenza del risultato ovvero dell'algoritmo, proprio a causa di questi errori. Questo problema non riguarderà comunque nessuna delle applicazioni discusse in questo testo.

<sup>70</sup> Chi è abituato a programmare in C++, a dichiarare le variabili in prossimità del proprio utilizzo, deve momentaneamente perdere tale buona abitudine, da non perdere quando si scrive in C++.

<sup>71</sup> Questi tipi di variabili vengono spesso utilizzati per la realizzazione di buffer di lettura e scrittura.

```
}
```

Nell'esempio appena scritto si è dichiarato un Array di caratteri di 10 elementi. Per richiamare un elemento di un Array è sufficiente scrivere l'indice dell'elemento che si vuole richiamare all'interno delle parentesi quadre. Ciò rimane valido anche se si vuole scrivere all'interno di un elemento dell'Array stesso. Quanto detto fino ad ora nasconde però qualcosa di pericoloso...se non noto. Si è affermato che mioArray è un Array di 10 interi il primo elemento è però mioArray[0] mentre l'ultimo è mioArray[9] e non mioArray[10] come si potrebbe pensare. Altro rischio quando si lavora con gli Array è che bisogna essere sempre certi che il programma non vada a leggere indici maggiori di 9. Infatti sarà possibile leggere anche mioArray[28] ma questo elemento non è in realtà parte del nostro Array<sup>72</sup>. Come esempio riportiamo questo segmento di programma:

```
void main (void) {  
  
    // Dichiarazione Array di caratteri con 10 elementi  
    char mioArray[10];  
  
    // Scrivo 23 nel primo elemento dell'Array  
    mioArray[0] = 23;  
  
    // Copio l'elemento 0 nell'elemento 2  
    mioArray[2] = mioArray[0];  
}
```

Si fa presente che l'Array, una volta dichiarato, possiede al suo interno valori casuali; sarà compito del programmatore aver cura d'inizializzarlo. L'inizializzazione di un Array può avvenire in vari modi. Un modo è quello di inizializzare un valore alla volta richiamando un indice alla volta. Un secondo metodo più pratico è quello di inizializzare l'Array durante la sua dichiarazione, la sintassi è la seguente:

```
char miaStringa [] = "ciao";  
int mioArray [] = {1,2,3,4,5};
```

In questo esempio si crea una stringa, ovvero un insieme di caratteri. Come si vedrà in seguito una stringa è caratterizzata dal fatto che deve terminare con un carattere speciale /0. Nel caso in cui la si inizializzi come sopra, il compilatore inserirà automaticamente il simbolo di fine stringa. Dunque anche se apparentemente l'Array sembra lungo 4 caratteri, ovvero miaStringa[0]-miaStringa[3], in realtà l'Array è lungo 5 celle, ovvero fino a miaStringa[4]. In particolare si sarà notato che la lunghezza dell'Array non è stata definita, infatti si sono lasciate le parentesi quadre senza dimensioni. Il compilatore, a seconda del valore d'inizializzazione assegnerà le dimensioni opportune, in questo caso 5.

Per inizializzare un Array con dei numeri interi, o semplicemente con 0, si procede ponendo i numeri per l'inizializzazione all'interno di parentesi graffe. Anche in questo caso non si è scritta la dimensione dell'Array. Il compilatore, a seconda del valore dei termini d'inizializzazione creerà un Array di dimensioni opportune. Nel caso di Array bidimensionali l'inizializzazione è fatta nel seguente modo:

<sup>72</sup> Errori tipo questi sono difficili da individuare e possono creare comportamenti inaspettati solo quando l'indice va oltre il range consentito. Questo potrebbe avvenire durante la fase di sviluppo, mostrando dunque la presenza del problema, ma potrebbe nel caso peggiore mostrarsi solo quando abbiamo messo il nostro sistema nelle mani del cliente. Nella programmazione su PC tale tipo di problema è noto come *buffer overflow* e viene sfruttato dai virus per accedere locazioni di memoria altrimenti inaccessibili. Questo problema potrebbe anche permettere ad un semplice file di testo di inglobare un virus...anche se apparentemente un file di testo non è un eseguibile!

```
int mioArray[][] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}};
```

In questo caso la variabile `mioArray` è una matrice 3 righe 4 colonne. In Figura 46 è riportata la finestra di Watch che si ottiene visualizzando le variabili appena inizializzate. Si noti in particolare la lunghezza della variabile `miaStringa`. Con l'introduzione degli Array si hanno tutti i tipi di variabili sufficienti per gestire ogni tipo di problema.

Oltre ai tipi di variabili ora descritte, si ha anche la possibilità di definirne di nuove, per mezzo delle quali è possibile gestire in maniera più snella strutture dati più complesse. Questo tipo di variabile va sotto il nome di struttura ovvero `struct`, la sua sintassi è del tutto simile all'ANSI C.

Per esempio, se dobbiamo risolvere un problema che gestisce rettangoli, sappiamo che un rettangolo avrà sempre un'altezza e una larghezza. Dal momento che questi parametri caratterizzano ogni rettangolo, è possibile dichiarare un nostro tipo di variabile chiamata rettangolo e che sia caratterizzata dai parametri precedentemente scritti, ovvero larghezza e altezza. Per dichiarare una variabile rettangolo si deve procedere come segue:

```
typedef struct {  
    unsigned char larghezza;  
    unsigned char altezza;  
} rettangolo;
```

Dunque bisogna scrivere `typedef struct` per poi scrivere all'interno delle parentesi graffe tutti i campi che caratterizzano la nostra variabile. Le variabili interne devono essere tipi primitivi, come `int`, `char`, o tipi che abbiamo precedentemente dichiarato con un'altra struttura. Alla fine delle parentesi graffe va scritto il nome della nostra struttura, ovvero il tipo della nostra variabile. Una volta creato il nostro tipo possiamo utilizzarlo per dichiarare delle variabili come si farebbe per una variabile intera. Vediamo il seguente esempio:

```
#include <p18f4550.h>  
  
#pragma config FOSC = HS  
#pragma config WDT = OFF  
#pragma config LVP = OFF  
#pragma config PBADEN = OFF  
  
//OSC = HS          Impostato per lavorare ad alta frequenza  
//WDT = OFF         Disabilitato il watchdog timer  
//LVP = OFF         Disabilitato programmazione LVP  
//PBADEN = OFF      Disabilitato gli ingressi analogici  
  
// Dichiarazione della struttura rettangolo  
typedef struct {  
    unsigned char larghezza;  
    unsigned char altezza;  
} rettangolo;  
  
void main (void) {
```

```

// Dichiarazione della variabile di tipo rettangolo
rettangolo figura;

// Imposto PORTA tutti ingressi
LATA = 0x00;
TRISA = 0xFF;

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

// Inizializzazione del record
figura.altezza = 10;
figura.larghezza = 3;

// Scrivo su PORTD l'altezza
LATD = figura.altezza;

// Ciclo infinito
while (1) {

}
}

```

E' possibile vedere che la dichiarazione del nostro tipo è stata fatta fuori dalla funzione main, questo non è obbligatorio ma potrebbe essere utile poiché in questo modo posso utilizzare questa dichiarazione anche per altre funzioni e non solo nella funzione main<sup>73</sup>. Per quanto riguarda la spiegazione, essendo il programma del tutto simile al progetto Hello\_World, non mi soffermerò sui dettagli. E' bene notare che la sintassi del C non permette di inizializzare le variabili interne alla struttura nella fase della loro dichiarazione. L'inizializzazione delle variabili definite all'interno della struttura deve essere fatta un campo alla volta. Dalle seguenti righe di codice è possibile notare che la dichiarazione di una variabile facendo uso di una struttura è del tutto analoga alla dichiarazione di una variabile di tipo standard. Si possono anche creare degli Array di tipo rettangolo.

```

// Dichiarazione della variabile di tipo rettangolo
rettangolo figura;

```

Dal seguente codice è possibile vedere che per accedere ai campi della nostra variabile figura bisogna utilizzare il punto. Da quanto si è detto si capisce quanto precedentemente spiegato sulla porta LATxbits, ovvero che dietro tale variabile è presente una struttura.

```

// Inizializzazione del record
figura.altezza = 10;
figura.larghezza = 3;

```

<sup>73</sup> Per ulteriori informazioni a riguardo si rimanda al paragrafo sulla visibilità delle variabili (scope).

---

In ultimo il valore della variabile viene scritta sulla PORTD.

```
// Scrivo su PORTD l'altezza  
LATD = figura.altezza;
```

Dal codice scritto si capisce come, facendo uso di commenti ben posti e di un nome di variabile adeguatamente chiaro, sia possibile leggere il codice come se stessi leggendo un libro. Un buon codice deve infatti essere “auto-spiegante” senza necessitare di un commento per ogni linea di codice<sup>74</sup>. Per poter caricare il programma su Freedom II, bisogna disabilitare il display LCD ed abilitare la stringa LED. Le impostazioni sono le stesse utilizzate per il progetto Hello\_World.

Con la struttura si ha a disposizione ogni tipo di variabile per affrontare qualunque problema. Si fa presente che in C non è presente il tipo stringa; la stringa in C viene realizzata per mezzo di un Array di caratteri, dunque un insieme di caratteri rappresenta una stringa. La particolarità delle stringhe è che l'ultimo carattere della stringa deve essere il carattere speciale '\0'. Dunque se si ha un Array di 10 elementi e si volesse scrivere Freedom, all'elemento 7 dell'Array bisogna caricare il carattere speciale '\0' di fine stringa, che in questo caso è più corta di dieci elementi. Nel dichiarare un Array che conterrà una stringa bisognerà sempre aggiungere un elemento rispetto al nome o frase più lunga, in modo da poter inserire il carattere speciale anche nel caso di frase di lunghezza massima. Per manipolare le stringhe, il C18 fornisce la libreria `string.h` che dovrà essere inclusa con la direttiva `#include`, ovvero `#include <string.h>`. Ulteriori dettagli sulle stringhe verranno dati nel paragrafo in cui si parlerà su come utilizzare un display alfanumerico LCD.

Alcune volte si ha l'esigenza di avere una specie di variabile che conterrà lo stesso valore durante tutto il programma. Questo tipo di variabile è più propriamente detta costante poiché a differenza di una variabile non è possibile variare il suo valore per mezzo del programma in esecuzione, ovvero è possibile cambiare il loro valore solo prima della compilazione.

Normalmente il nome delle costanti viene scritto in maiuscolo ma questa è solo una convenzione. In C per poter definire una costante si usa la direttiva `#define`<sup>75</sup> per mezzo della quale si definisce una corrispondenza tra un nome e un valore numerico. In questo modo nel programma ogni volta che bisognerà scrivere questo valore basterà scrivere il nome che gli si è assegnato. Questo è un tipico esempio:

```
#define MAX_VALUE 56
```

Si capisce che l'utilizzo delle costanti risulta particolarmente utile qualora si vogliano definire dei limiti. L'utilità dell'aver definito la costante `MAX_VALUE` è che se si dovesse variare questo valore non sarà necessario cambiarlo in ogni punto del programma ma basterà cambiare la riga precedente con il nuovo valore.

L'utilizzo di numeri nel codice è noto come utilizzo di numeri magici (*magic number*) il suo utilizzo è altamente sconsigliato, salvo in codici brevi di esempio. Infatti quando ci si trova a cambiare dei *magic number*, è facile cambiare un numero per un altro, creando

---

<sup>74</sup> In questo testo si fa uso di più commenti di quanto non siano in realtà necessari.

<sup>75</sup> La direttiva `#define` può essere utilizzata anche per la definizione di macro ovvero un blocco di codice che è possibile riscrivere facendo riferimento al nome con cui è stato definito. Una macro non è una funzione poiché al posto del nome da dove viene richiamata viene sostituito l'intero codice e non un salto. Il suo utilizzo viene sconsigliato in C++ poiché può portare a problemi difficili da trovare. Personalmente consiglio di realizzare una funzione ogni qual volta si stia pensando di realizzare una macro. Eccezioni sono sempre possibili, ma state in guardia.

---

problemi al codice. Un esempio di utilizzo della direttiva `#define` può essere il seguente. Si vede subito la sua utilità qualora il buffer dovesse avere dimensioni diverse.

```
#define BUFFER_SIZE 10

void main (void) {

    // Dichiarazione Array di caratteri con BUFFER_SIZE elementi
    char mioArray[BUFFER_SIZE];

    // ...resto del codice

}
```

La direttiva `#define` oltre che a definire costanti e macro risulta molto utile per assegnare un nome ad un determinato pin del PIC. In questo modo se il pin viene spesso utilizzato nel programma si può far riferimento al nome piuttosto che al nome del pin stesso. Questo metodo di rinominare il pin risulta particolarmente utile qualora in fase di sviluppo si voglia cambiare il pin dove è per esempio attaccato un LED rosso di allarme. Infatti basterà in un solo punto del programma cambiare l'assegnazione di un nome ed il resto del programma farà uso del nuovo pin senza problemi. Un esempio di assegnazione di un nome ad un pin è il seguente:

```
#define LED_ROSSO LATDbits.LATD1
```

o facendo uso del registro `PORTx`:

```
#define LED_ROSSO PORTDbits.RD1
```

Nel caso in cui si voglia dichiarare una famiglia di costanti, la direttiva `#define` può essere utilizzata per dichiarare tutte le costanti. Per esempio se si volesse scrivere un valore di inizializzazione particolare per i valori contenuti in una struttura del tipo rettangolo, potremmo scrivere:

```
#define BASE_STANDARD 10
#define ALTEZZA_STANDARD 20
```

ed inizializzare il nostro record o Array con tali valori. Un altro modo utilizzato in C per raggruppare delle costanti è per mezzo della struttura `enum`, ovvero enumerando le costanti da utilizzare. La sintassi per definire una struttura `enum` è molto simile a quella utilizzata per definire un nuovo tipo di variabile.

```
enum RETTANGOLO_STANDARD {

    BASE_STANDARD          = 10,
    ALTEZZA_STANDARD       = 20

};
```

Si può vedere che dopo `enum` bisogna mettere il nome della raccolta di costanti. Le costanti diversamente dalla direttiva `#define` vengono assegnate ad un valore per mezzo dell'operatore `=`, inoltre al termine della riga viene posta una virgola piuttosto che un punto e virgola, mentre l'ultimo elemento, diversamente dagli altri non ha la virgola. La costante definita all'interno di una struttura `enum` può essere utilizzata allo stesso modo di una costante



---

definita per mezzo della direttiva `#define` ovvero senza il punto utilizzato normalmente per le strutture dati.

Come ultima nota si ricorda che dal momento che il C è case sensitive, ovvero distingue le maiuscole dalle minuscole, la variabile dichiarata come `mioNumero` è diversa dalla variabile `MioNumero`, anche se sono dello stesso tipo. Il compilatore inoltre non permette di definire all'interno dello stesso scope variabili con lo stesso nome, anche se di tipo differente.

---

## Operatori matematici

Ogni volta che si crea una variabile si ha più o meno l'esigenza di manipolare il suo contenuto per mezzo di operazioni algebriche. Il C18 supporta in maniera nativa i seguenti operatori matematici:

+ : operatore somma  
- : operatore sottrazione  
/ : operatore divisione  
\* : operatore moltiplicazione  
% : resto divisione tra interi

Per operatori nativi si intende che, senza aggiunta di librerie esterne è possibile far uso di tali operatori. A questi operatori si aggiungono in realtà altre funzioni matematiche particolari, quali i  $\sin(x)$ ,  $\cos(x)$ ,  $\log(x)$ ..., e relative funzioni inverse. Per poter però utilizzare questi ulteriori operatori bisogna includere, per mezzo della direttiva `#include`, la libreria `math.h`<sup>76</sup>. Come possibile operatore di somma alcune volte viene utilizzato il doppio ++, che ha lo scopo di incrementare la variabile di uno. Consideriamo il seguente segmento di codice:

```
void main (void) {  
  
    int i=0;  
  
    // Dopo la somma i vale 1  
    i = i + 1;  
  
}
```

Un altro modo per effettuare questo tipo di somma in maniera snella è per mezzo dell'operatore incremento ++, come riportato nel seguente codice:

```
void main (void) {  
  
    int i=0;  
  
    // Dopo l'incremento i vale 1  
    i++;  
  
}
```

In maniera analoga all'operatore ++ esiste anche l'operatore di decremento --. Un esempio è riportato nel seguente codice.

```
void main (void) {  
  
    int i=0;  
  
    // Dopo l'incremento i vale 1  
    i++;  
  
    // Dopo il decremento i vale 0  
    i--;  
  
}
```

---

<sup>76</sup> Per ulteriori informazioni su tale libreria si rimanda alla documentazione ufficiale della Microchip che è possibile trovare nella directory doc della cartella principale dove è stato installato il C18.

Gli operatori d'incremento e decremento vengono utilizzati per ottimizzare il codice assembler durante la fase di compilazione. Infatti il microcontrollore possiede come istruzioni base l'operazione d'incremento e decremento di un registro senza far uso del registro speciale accumulatore. Analizziamo il seguente codice:

```
void main (void) {  
  
    int i = 0;  
    char y = 0;  
  
    // Sommo 1 ad un intero  
    i = i + 1;  
  
    // Incremento di un intero  
    i++;  
  
    // Sommo 1 ad un char  
    y = y + 1;  
  
    // Incremento di un char  
    y++;  
  
}
```

Richiamando dal menù *View* il *Disassembly Listing*, è possibile vedere come vengono tradotte in assembler le istruzioni di somma ed incremento del nostro esempio:

```
39:                                     // Sommo 1 ad un intero  
40:                                     i = i + 1;  
007C 0E01 MOVLW 0x1  
007E 6EE7 MOVWF 0xfe7, ACCESS  
0080 0E09 MOVLW 0x9  
0082 CFDB MOVFF 0xfdb, 0  
0084 F000 NOP  
0086 0E0A MOVLW 0xa  
0088 CFDB MOVFF 0xfdb, 0x1  
008A F001 NOP  
008C 50E7 MOVF 0xfe7, W, ACCESS  
008E 2600 ADDWF 0, F, ACCESS  
0090 0E00 MOVLW 0  
0092 2201 ADDWFC 0x1, F, ACCESS  
0094 0E09 MOVLW 0x9  
0096 C000 MOVFF 0, 0xfdb  
0098 FFDB NOP  
009A 0E0A MOVLW 0xa  
009C C001 MOVFF 0x1, 0xfdb  
009E FFDB NOP  
41:  
42:                                     // Incremento di un intero  
43:                                     i++;  
00A0 0E09 MOVLW 0x9  
00A2 2ADB INCF 0xfdb, F, ACCESS  
00A4 0E0A MOVLW 0xa  
00A6 E301 BNC 0xaa  
00A8 2ADB INCF 0xfdb, F, ACCESS  
44:  
45:                                     // Sommo 1 ad un char  
46:                                     y = y + 1;  
00AA 0E0B MOVLW 0xb
```

```

00AC    28DB    INCF 0xfdb, W, ACCESS
00AE    6EE7    MOVWF 0xfe7, ACCESS
00B0    0E0B    MOVLW 0xb
00B2    CFE7    MOVFF 0xfe7, 0xfdb
00B4    FFDB    NOP
47:
48:                // Incremento di un char
49:                y++;
00B6    2ADB    INCF 0xfdb, F, ACCESS

```

E' possibile subito vedere che il caso peggiore si ha quando si deve sommare 1 ad un intero, ovvero  $i = i + 1$ . Questo come spiegato è legato al fatto che un intero è rappresentato da due byte, dunque la sua gestione è più laboriosa del caso di un incremento di un byte. La cosa migliora nel caso in cui invece di sommare 1 si fa uso dell'operatore d'incremento `i++`, riducendo il codice assembler ad oltre la metà.

Qualora la nostra variabile sia un `char` invece di un intero, si ha che sommare un numero è più o meno costoso quanto il caso d'incremento di un intero, però quando incremento un `char` per mezzo dell'operatore `++` si ha che il codice assembler è ottenuto per mezzo di una sola istruzione, ovvero l'incremento diretto del registro! L'analisi appena svolta non è un ragionamento da prendere come verità assoluta, ovvero che l'incremento di uno genera sempre un codice assembler migliore della somma di 1. A seconda del compilatore utilizzato, e questo è vero per il C18 come per ogni altro linguaggio e ambiente di sviluppo, il codice che si ottiene può essere diverso. Un compilatore diverso si sarebbe potuto accorgere che `i` ed `y` venivano solo incrementate di 1, dunque le avrebbe potute gestire con un incremento di variabile, piuttosto che come somma di 1 ad una variabile. Quanto detto dovrebbe anche aprire gli occhi in termini di ottimizzazione, ovvero che per ottimizzare un codice non basta cambiare il sorgente solo per sentito dire, ma bisogna sempre controllare il codice assembler per vedere come viene tradotto il codice sorgente.

Detto questo cerchiamo di complicarci la vita dicendo che l'operatore `++` può essere messo sia prima che dopo la variabile, ovvero potevamo scrivere `y++` e `++y`, ottenendo risultati diversi! Vediamo un esempio:

```

// Inizializzo le variabili
charA = 0;
charB = 0;

// Incremento dopo la variabile
charB = charA++ + 1;

// Inizializzo le variabili
charA = 0;
charB = 0;

// Incremento prima della variabile
charB = ++charA + 1;

```

Come visibile, escluse le inizializzazioni delle variabili `charA` e `charB`, quello che si ha è che `charB` è data dalla somma di 1 più l'incremento di `charA`, una volta ottenuto con `++` posto prima della variabile, e una volta posto dopo. Ovviamente il risultato deve essere uguale a 2 in tutti e due i casi!

...ed è qui che avvengono i problemi!

Nel primo caso, `charB` varrà 1 mentre nel secondo caso `charB` vale 2!

La ragione è legata al fatto che il ++ prima o dopo effettivamente permette d'incrementare la variabile subito (ovvero prima della somma) o dopo la somma. Questo se non tenuto in considerazione può portare a gravi errori. Un modo semplice per evitare tali errori è far uso di una sola modalità d'incremento ++i o i++, una vale l'altra, e scrivere l'istruzione in una riga di codice a parte, ovvero non all'interno di espressioni di alcun tipo. In questo modo si garantisce che quello che ci si aspetta sia quello che si ottiene. Dal codice assembler è possibile vedere quanto ora spiegato. Questo esempio può essere un'ottima esercitazione nell'utilizzo del simulatore software, in particolare si consiglia di impostare la finestra Watch con i registri charA, charB e WREG (ovvero l'accumulatore).

```

51:                                     // Inizializzo le variabili
52:                                     charA = 0;
   00B8      6ADF      CLRF 0xfdf, ACCESS
53:                                     charB = 0;
   00BA      0E01      MOVLW 0x1
   00BC      6ADB      CLRF 0xfdb, ACCESS
54:
55:                                     // Incremento dopo la variabile
56:                                     charB = charA++ + 1;
   00BE      50DF      MOVF 0xfdf, W, ACCESS
   00C0      2ADF      INCF 0xfdf, F, ACCESS
   00C2      0F01      ADDLW 0x1
   00C4      6EE7      MOVWF 0xfe7, ACCESS
   00C6      0E01      MOVLW 0x1
   00C8      CFE7      MOVFF 0xfe7, 0xfdb
   00CA      FFDB      NOP
57:
58:                                     // Inizializzo le variabili
59:                                     charA = 0;
   00CC      6ADF      CLRF 0xfdf, ACCESS
60:                                     charB = 0;
   00CE      6ADB      CLRF 0xfdb, ACCESS
61:
62:                                     // Incremento prima della variabile
63:                                     charB = ++charA + 1;
   00D0      2ADF      INCF 0xfdf, F, ACCESS
   00D2      28DF      INCF 0xfdf, W, ACCESS
   00D4      6EE7      MOVWF 0xfe7, ACCESS
   00D6      0E01      MOVLW 0x1
   00D8      CFE7      MOVFF 0xfe7, 0xfdb
   00DA      FFDB      NOP

```

Dal codice assembler, in particolare dalle inizializzazioni, si può vedere che charA è il registro 0xFDF mentre charB è il registro 0xFDB. Nella prima somma è possibile vedere che il registro 0xFDF, ovvero charA, viene caricato in W prima del suo incremento (per mezzo dell'istruzione `MOVF 0xfdf, W, ACCESS`), dopo di che viene incrementato charA ed effettuata la somma tra il vecchio valore di charA (ovvero 0) ed 1. Questa è la ragione per cui la somma finale vale 1 invece di 2! Osservando la seconda somma si osserva invece che viene effettuato prima l'incremento e solo successivamente il valore di charA viene caricato in W per effettuare la somma con 1. Dunque il valore finale varrà in questo caso 2.

Una situazione di questo tipo, come detto va evitata poiché basta poca distrazione per non pensare al problema che si può avere dietro. Un altro caso in cui l'operatore d'incremento potrebbe creare problemi è all'interno dell'istruzione condizionali `if ()` che verrà illustrata a breve.

---

Dopo questa “logorroica” spiegazione dell'operatore somma ed incremento, vediamo qualche dettaglio sugli operatori rimanenti.

L'operazione di divisione e moltiplicazione sono operazioni molto complesse e per la loro esecuzione richiedono molto tempo e memoria del PIC. In alcuni casi queste operazioni possono essere sostituite con shift a destra (divisione) o shift a sinistra (moltiplicazione) ma solo qualora l'operazione sia una potenza di due. Alcuni esempi verranno riportati a breve quando si introdurranno gli operatori bitwise.

I PIC18 diversamente dai suoi predecessori ad 8 bit, possiede al suo interno un moltiplicatore hardware 8x8 per mezzo del quale è possibile ottimizzare il codice per lo svolgimento delle moltiplicazioni. In particolare una moltiplicazione tra due byte può essere svolta in un solo ciclo istruzione. Naturalmente questi dettagli non sono molto importanti al nostro livello, ma è bello pensare che il compilatore farà uso di questo hardware per ottimizzare il codice per risolvere le moltiplicazioni e divisioni.

Per quanto riguarda la divisione è bene notare che, salvo casi specifici l'operazione deve essere svolta tra variabili di tipo float. Se le variabili non dovessero essere di tipo float qualcosa di strano avverrà ma verrà descritto nel prossimo paragrafo, dove verrà introdotto il concetto di casting delle variabili.

L'unico operatore strano tra quelli citati (trascurando il trauma per gli incrementi, che si era pensato di gestire facilmente), è rappresentato dall'operatore % che ritorna il resto della divisione tra interi, ovvero sia il dividendo che il divisore devono essere interi. Non c'è nulla di complicato dietro questo simbolo infatti gli esempi spiegheranno il tutto.

```
intA = 0;
intB = 10;

// Divisione tra interi
intA = intB % 5;

intA = 0;

// Divisione tra interi
intA = intB % 3;
```

Nel primo caso, dal momento che intB vale dieci e viene diviso per un suo sottomultiplo, si avrà che il resto sarà 0, dunque intA vale 0. Nel secondo caso dal momento che intB vale 10 e viene diviso per 3, si avrà resto 1, dunque intA varrà 1. Se si fa utilizzo dell'operatore % tra variabili non intere si avrà errore di compilazione.

Ultima nota va fatta riguardo alla priorità degli operatori. Questa è la stessa nota dalla algebra delle elementari, ma al fine di evitare problemi, visto la presenza anche di operatori “strani”, piuttosto che scrivere e ricordarsi il loro ordine di priorità (cosa che comunque dimenticherete!) è bene far uso delle parentesi tonde in modo da raggruppare ed ordinare espressioni complesse.

```
a = b * c + 5;
```

è meglio scriverla:

```
a = (b * c) + 5;
```

---

piuttosto che far affidamento al fatto che la moltiplicazione viene svolta comunque per prima. Nel caso in cui si voglia fare prima la somma sarà invece obbligatorio scrivere:

```
a = b * (c + 5);
```

Nel caso di espressioni complesse si raccomanda inoltre di creare delle variabili tampone intermedie, il cui nome deve essere espressivo della loro funzione. Per esempio per il calcolo della velocità della bicicletta l'espressione potrebbe essere:

```
speed = (numero_giri_ruota * DIAMETRO_RUOTA) / (num_base_tempo * BASE_TEMPO);
```

potrebbe essere scritta meglio come:

```
spazio_percorso = numero_giri_ruota * DIAMETRO_RUOTA;
tempo_percorso = num_base_tempo * BASE_TEMPO;
speed = spazio_percorso / tempo_percorso;
```

Si noti che in questo caso il tutto risulta molto più leggibile, questo risulterà particolarmente pratico nel caso in cui si debbano risolvere dei problemi<sup>77</sup>. Dalle formule e dal nome delle variabili ci si accorge che in realtà l'operazione è svolta facendo uso anche di alcune costanti. Il modo con cui vengono scritte le variabili e costanti è molto utile, poiché permette facilmente di capire se si ha a che fare con costanti o variabili. Come detto le costanti sono tutte maiuscole, mentre le variabili hanno la caratteristica di essere scritte in minuscolo. Altra pratica in nomi composti è quella di usare o un *under score* \_ per unire i nomi o scrivere il secondo nome con l'iniziale maiuscola.

```
tempoPercorso = numBaseTempo * BASE_TEMPO;
```

Questa pratica per nominare le variabili è anche seguita per nominare le funzioni. In ultimo si noti l'importanza di dare alle variabili un nome significativo. Questo eviterà di ridurre i commenti mantenendo il codice più pulito ma sempre molto leggibile.

---

<sup>77</sup> Questa soluzione anche se più leggibile ha il costo di più variabili, ma se non si hanno troppe restrizioni è un buon prezzo da pagare.

---

## Casting

Per casting si intende una promozione o un cambio di un tipo di una variabile in un altro tipo. La sua utilità e pericoli a cui può condurre sono difficilmente spiegabili in parole, dunque spiegherò il tutto tramite esempi. Come inizio, al fine di tranquillizzare gli animi inquieti, sappiate che qualora svolgiate operazioni tra variabili tutte dello stesso tipo le sorprese sono poche...ma il rischio di perdere informazioni è sempre in agguato! Prendiamo in considerazione il seguente esempio:

```
intA = 9 / 4;  
floatA = 9 / 4;
```

Quando si scrive un numero come riportato nell'esempio, il compilatore lo considera come intero, dunque la prima operazione è una divisione tra interi e il risultato è intero. Se facessimo  $9/4$  con la calcolatrice (meglio se a mente) viene 2.25, però dal momento che il nostro risultato è memorizzato in un intero, il fatto che la variabile `intA` valga 2 non sorprende molto, infatti il risultato viene apparentemente troncato. Quello che non ci si aspetta (e qui state già capendo che sto per dire qualcosa di strano) è che pur mettendo il risultato in una variabile di tipo `float`, questo sarà ancora 2!

Il problema infatti non sta sul valore sinistro (come viene spesso nominato in inglese *left value* o *lvalue*), ma su quello destro. Il compilatore quando deve svolgere l'operazione di divisione prende le due variabili di tipo `int`, ovvero a 16 bit, e svolge l'operazione. Per come sono strutturati i numeri ovvero interi, la parte decimale è già persa, dunque quello che viene caricato nella variabile `floatA` è effettivamente il risultato della divisione tra interi.

Per evitare che la divisione dia risultato 2 anche quando la variabile per il risultato è di tipo `float`, bisogna promuovere, ovvero fare il casting di uno dei due numeri del valore destro (o del dividendo o del divisore), ovvero:

```
floatA = (float) 9 / 4;
```

Dall'esempio si vede subito che fare il casting significa mettere tra parentesi il valore a cui vogliamo convertire la nostra variabile (o costante in questo caso). A questo punto il compilatore quando troverà l'operazione vedrà che la divisione è tra un `float` ed un `int`, dunque nel fare la divisione, al fine di gestire i calcoli, utilizzerà la struttura dati più complessa ovvero `float`.

Lo stesso risultato si sarebbe ottenuto qualora nel fare la divisione, o il dividendo o il divisore sia già di tipo `float`:

```
float floatA = 9;  
floatB = floatA / 4;
```

In questo caso infatti `floatB` varrà 2.25 come nel caso in cui era stata fatta la promozione volontaria a `float`. Alcune volte quando tutte le variabili sono di tipo `float` e si sta facendo una divisione di cui non si ha interesse nella parte decimale, ovvero si vuole solo la parte intera della divisione, può ritornare utile fare un casting ad `int`:

```
float floatA = 9;  
floatB = (int) floatA / 4;
```



---

In questo caso `floatB`, grazie al casting, ci permette di avere come risultato 2. Dunque non tutti i mali vengono per nuocere, ma bisogna certamente stare attenti. Da quanto spiegato sembra che il pericolo stia solamente nell'utilizzo dei float e con la divisione, mentre in realtà ogni volta che facciamo operazioni con strutture dati differenti, ovvero tipi diversi, c'è il rischio che a causa di una qualunque manipolazione delle variabili ci siano delle perdite d'informazione. Vediamo ora un esempio tra `char` e `int`.

```
intA = 10;

charA = intA;

intA = 300;

charA = intA;
```

Nel primo caso carichiamo 10 in una variabile intera e poi carichiamo tale variabile `int` in una `char`. Quello che ci si aspetta è che `charA` valga 10. Effettivamente questo è il valore di `charA`. A questo punto sembra che tale operazione possa essere fatta in maniera innocua, però ripetendo l'operazione con il valore di 300 vediamo che `charA` vale questa volta 44!

Effettivamente 300 in binario si scrive 100101100, ovvero richiede due byte, cosa che un `int` effettivamente ha. Nel momento in cui carico il valore in un `char` solo il primo byte viene caricato dunque solo i primi 8 bit, ovvero 00101100, cioè il nostro 44. In questo caso abbiamo dunque perso delle informazioni ma il casting non ci avrebbe aiutato. Quello che avremo dovuto fare era quello di mettere il risultato in un intero invece che in un `char`.

Questo secondo esempio serve anche per mettere in guardia nel caso in cui si facciano operazioni complesse in cui valori intermedi possono causare l'overflow di qualche variabile.

---

## Operatori logici e bitwise

Gli operatori logici rappresentano quegli operatori che permettono “al programma” di rispondere a delle domande con una risposta positiva, ovvero 1 logico, o negativa, ovvero 0 logico. Tali operatori sono:

`||` : operatore logico OR  
`&&` : operatore logico AND  
`==` : operatore logico di uguaglianza  
`!=` : operatore logico diverso  
`<=` : operatore logico minore o uguale  
`>=` : operatore logico maggiore o uguale  
`>` : operatore logico maggiore  
`<` : operatore logico minore

Tali operatori verranno discussi in maggior dettaglio quando si parlerà delle istruzioni condizionali, ovvero quelle istruzioni che permettono al programma di gestire le domande poste per mezzo degli operatori logici. Come verrà messo in evidenza parlando dell'istruzione `if (...)` la domanda *A* è uguale a *B* si pone scrivendo `if (A==B)` e non `if (A=B)`.

Gli operatori bitwise permettono di manipolare un registro (variabile) variandone i singoli bit. Tali operatori sono quelli che permettono di svolgere operazioni secondo l'algebra booleana. Gli operatori bitwise sono:

`&` : operatore binario AND  
`|` : operatore binario OR  
`^` : operatore binario XOR  
`~` : operatore complemento a 1 (i bit vengono invertiti)<sup>78</sup>  
`<<` : shift a sinistra  
`>>` : shift a destra

Come visibile *and* ed *or* binario sono molto simili a quelli logici, se non per il fatto che il simbolo deve essere ripetuto solo una volta. Gli operatori di shift intervengono sulla variabile con uno spostamento di un bit verso sinistra o verso destra dei bit che compongono il valore numerico originale. Il bit che viene inserito è uno zero mentre il bit che esce viene perduto. Spostare verso sinistra di un bit equivale a moltiplicare per due, mentre spostare verso destra di un bit equivale a dividere per due. Si capisce dunque che se l'operazione di divisione o moltiplicazione deve essere fatta per una potenza di due è bene utilizzare gli operatori di shift visto che richiedono ognuno un solo ciclo istruzione. In questo modo si riesce a risparmiare tempo e spazio in memoria. Vediamo un esempio estremo in cui si voglia dividere per 4 una variabile di tipo `char`:

```
charA = 16;

// Divisione classica
charB = charA /4;

charA = 16;

// Divisione per mezzo dello shift
```

---

<sup>78</sup> Il simbolo `~` è possibile inserirlo come carattere ASCII numero 126. Tenere premuto ALT, digitare 126 e poi rilasciare ALT.

```
charB = charA >> 2;
```

Dal momento che la divisione è per 4 ovvero una potenza di due è possibile utilizzare lo shift a destra, di due posizioni, quale passo equivalente alla divisione per due. Controllando il codice assembler è possibile vedere che effettivamente lo shift ha permesso di rendere l'implementazione della divisione più snella, permettendo di raggiungere dunque maggiori velocità di esecuzione e risparmiare memoria.

```
26:                                charA = 16;
    0282    0E10    MOVLW 0x10
    0284    6EF3    MOVWF 0xff3, ACCESS
    0286    0E08    MOVLW 0x8
    0288    CFF3    MOVFF 0xff3, 0xfdb
    028A    FFDB    NOP
27:
28:                                // Divisione classica
29:                                charB = charA /4;
    028C    0E04    MOVLW 0x4
    028E    6E14    MOVWF 0x14, ACCESS
    0290    C014    MOVFF 0x14, 0xe
    0292    F00E    NOP
    0294    0E08    MOVLW 0x8
    0296    CFDB    MOVFF 0xfdb, 0x9
    0298    F009    NOP
    029A    EC6C    CALL 0xd8, 0
    029C    F000    NOP
    029E    0E09    MOVLW 0x9
    02A0    C009    MOVFF 0x9, 0xfdb
    02A2    FFDB    NOP
30:
31:                                charA = 16;
    02A4    0E10    MOVLW 0x10
    02A6    6EF3    MOVWF 0xff3, ACCESS
    02A8    0E08    MOVLW 0x8
    02AA    CFF3    MOVFF 0xff3, 0xfdb
    02AC    FFDB    NOP
32:
33:                                // Divisione per mezzo dello shift
34:                                charB = charA >> 2;
    02AE    40DB    RRNCF 0xfdb, W, ACCESS
    02B0    40E8    RRNCF 0xfe8, W, ACCESS
    02B2    0B3F    ANDLW 0x3f
    02B4    6EE7    MOVWF 0xfe7, ACCESS
    02B6    0E09    MOVLW 0x9
    02B8    CFE7    MOVFF 0xfe7, 0xfdb
    02BA    FFDB    NOP
```

Dal codice assembler è possibile vedere che per fare la divisione per 4 si è effettuato uno shift a destra di due posti (ovvero eseguita `RRNCF` per due volte) ottenendo come risultato 2. Se ripetete l'esempio dividendo per 16 invece di 4 vi accorgete di una cosa strana, l'operazione di shift non viene tradotta più facendo uso dell'istruzione `RRNCF` :

```
48:                                // Divisione per mezzo dello shift
49:                                charB = charA >> 4;
    02E8    38DB    SWAPF 0xfdb, W, ACCESS
    02EA    0B0F    ANDLW 0xf
    02EC    6EE7    MOVWF 0xfe7, ACCESS
    02EE    0E09    MOVLW 0x9
    02F0    CFE7    MOVFF 0xfe7, 0xfdb
```

Questo esempio dovrebbe mettervi in guardia ancora una volta che non bisogna mai ottimizzare il codice alla cieca, soprattutto quando sono presenti delle costanti. Ripetendo ancora una volta il nostro esempio facendo però uso di variabili di tipo float si ha:

```
floatA = 16;

// Divisione classica
floatB = floatA /4;

floatA = 16;

// Divisione per mezzo dello shift
floatB = (int) floatA >> 2;
```

In questo caso è necessario effettuare un casting della variabile float ad intero, poiché l'operatore shift può essere utilizzato solo per variabili di tipo intero o char. Controllando il codice assembler (qui non riportato per brevità) si può vedere che in ambe due le implementazioni sono necessarie ben 40 istruzioni. Dunque se si pensava di ottimizzare il codice per mezzo dell'utilizzo dello shift si sarebbe fatto un buco nell'acqua<sup>79</sup>. In casi come questo dove non si ha il beneficio voluto è bene far uso della forma del codice di più facile lettura, ovvero:

```
floatB = floatA /4;
```

piuttosto che della seconda:

```
floatB = (int) floatA >> 2;
```

Anche questo esempio dovrebbe essere una “lampadina” da utilizzare nel vostro bagaglio d'esperienza, nel caso in cui vogliate ottimizzare il codice.

Riprendiamo ora l'esempio discusso nel paragrafo del casting:

```
intA = 300;

charA = intA;
```

In tale esempio si era messo in evidenza che trasferire un intero troppo grande in un char avrebbe portato alla perdita dei dati. Supponiamo ora di voler evitare la perdita di dati, facendo uso di due variabili char:

```
intA = 300;

charA = intA;
charB = intA >> 8;
```

Grazie allo shift è possibile recuperare il byte che andava perduto e caricarlo in charB. Tale operazione risulta molto utile nel caso in cui si voglia per esempio memorizzare il valore di un intero in memoria EEPROM, visto che la EEPROM è composta da locazioni di memoria da un byte. Questo codice se pur funzionante non mette in evidenza il fatto che vi è una

<sup>79</sup> Compilatori differenti possono ottimizzare in maniera diversa, rendendo o meno l'utilizzo dello shift una soluzione valida per ottimizzare il codice.

---

trasformazione in `char` dunque è bene commentare tale operazione o far uso di un casting:

```
intA = 300;

charA = (char) intA;
charB = (char) (intA >> 8);
```

Fare un casting non renderà il codice assembler più complesso, ma metterà in evidenza che stiamo trasformando il nostro intero in `char`. Si noti che in questa linea di codice l'operazione di shift è messa tra parentesi.

```
charB = (char) (intA >> 8);
```

Se non si mettesse l'operazione tra parentesi il casting verrebbe applicato ad `intA` perdendo dunque l'informazione del secondo byte. Facendo poi lo shift di 8 posizioni di un `char` ovvero di un byte si otterrebbe 0 invece di 1. In ultimo visto che in questo paragrafo si sono messe in evidenza diverse ottimizzazioni, si noti che la traduzione in assembler dello shift di 8 posizioni è in realtà ottimizzato senza neanche utilizzare l'istruzione assembler di shift.

```
69:                                charB = (char) (intA >> 8);
03D4    0E0A    MOVLW 0xa
03D6    CFDB    MOVFF 0xfdb, 0x14
03D8    F014    NOP
03DA    0E0B    MOVLW 0xb
03DC    CFDB    MOVFF 0xfdb, 0x15
03DE    F015    NOP
03E0    C015    MOVFF 0x15, 0x14
03E2    F014    NOP
03E4    6A15    CLRF 0x15, ACCESS
03E6    0E09    MOVLW 0x9
03E8    C014    MOVFF 0x14, 0xfdb
03EA    FFDB    NOP
```

Questo discende proprio dal fatto che lo shift di 8 posizioni è frequentemente utilizzato come istruzione ed equivale a buttare via un byte, per tale ragione è ottimizzabile senza l'utilizzo dello shift. Ancora una volta controllate il codice assembler per ottimizzare il vostro codice!

---

## Il ciclo for ( )

I programmi difficilmente sono lineari, ovvero una semplice sequenza di istruzioni da compiere dall'inizio alla fine. Spesso l'esecuzione del programma viene interrotta per fare dei test o ripetere delle porzioni del programma stesso. Ripetere una sezione del programma equivale a fare dei salti all'indietro in modo da riposizionare il registro PC all'inizio del ciclo.

Per effettuare un ciclo l'istruzione più semplice e sicuramente la più usata è l'istruzione `for(...)`, la quale permette appunto di eseguire un numero definito di volte una certa operazione o insieme di operazioni. La sua sintassi è:

```
for (espressione1; espressione2; espressione3) {  
    // Blocco da ripetere  
}
```

Si può vedere che il ciclo `for` è composto da un blocco con parentesi graffe al cui interno vanno scritte l'insieme d'istruzioni da ripetere. Prima del blocco da ripetere, vi è la parola chiave `for` e tre espressioni tra parentesi<sup>80</sup>:

**espressione1:** Usata per l'inizializzazione della variabile di controllo loop.

**espressione2:** Controllo della condizione di fine loop.

**espressione3:** Operazione da svolgere sulla variabile per il controllo del loop.

Vediamo di capirci qualcosa con il seguente esempio :

```
#include <p18f4550.h>  
  
#pragma config FOSC = HS  
#pragma config WDT = OFF  
#pragma config LVP = OFF  
#pragma config PBADEN = OFF  
  
//OSC = HS          Impostato per lavorare ad alta frequenza  
//WDT = OFF         Disabilito il watchdog timer  
//LVP = OFF         Disabilito programmazione LVP  
//PBADEN = OFF      Disabilito gli ingressi analogici  
  
#define MAX_VALUE 86  
  
void main (void) {  
    // Variabile per il conteggio  
    unsigned char i;  
  
    // Imposto PORTA tutti ingressi  
    LATA = 0x00;  
    TRISA = 0xFF;  
  
    // Imposto PORTB tutti ingressi  
    LATB = 0x00;  
    TRISB = 0xFF;  
  
    // Imposto PORTC tutti ingressi
```

---

<sup>80</sup> Ogni espressione può essere in realtà un gruppo di espressioni, ma si sconsiglia di utilizzare tale tecnica poiché di più difficile lettura.

```

LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

for (i=0; i < MAX_VALUE; i++) {

    LATD = i;
}

// Ciclo infinito
while (1) {

}
}

```

La parte iniziale del programma non richiede particolari commenti. Si noti solo che si è voluto definire una costante `MAX_VALUE`, la quale verrà utilizzata come limite per il controllo del loop. Il suo utilizzo non è obbligatorio, ma ritorna utile in applicazioni reali, permettendo di cambiare il numero di volte che verrà ripetuto il loop. Si noti inoltre che `PORTD` è stata impostata come output, in modo da visualizzare il valore `i`.

Iniziamo a vedere in maggior dettaglio le espressioni del ciclo `for`. La prima espressione inizializza la variabile `i`, precedentemente dichiarata, questa verrà utilizzata per il conteggio delle volte che è stato ripetuto il loop. Normalmente le variabili di conteggio vengono inizializzate a 0. Se si hanno particolari esigenze è possibile iniziare il conteggio ad un valore differente. La seconda espressione effettua ad ogni ciclo un controllo sullo stato della variabile per vedere se la condizione è verificata. In questo caso volendo fare un ciclo con `MAX_VALUE` iterazioni si è posto `i = 0` come inizio e `i < MAX_VALUE` come fine.

La terza espressione del ciclo `for` è il tipo di conteggio che si vuole avere, in questo caso, partendo da 0 e volendo raggiungere `MAX_VALUE` bisogna incrementare di 1, dunque si è scritto `i++`. Se il punto di partenza fosse stato `MAX_VALUE` e quello di arrivo fosse stato `>0` si sarebbe dovuto scrivere `i--`<sup>81</sup>.

Le istruzioni che si vogliono ripetere per `MAX_VALUE` volte sono contenute all'interno delle parentesi graffe. In questo caso si ha la sola istruzione che scrive il valore di `i` sulla `PORTD` ovvero sui LED. Programmando la scheda di sviluppo Freedom II, precedentemente impostata con i Jumper nello stesso modo del progetto `Hello_World`, è possibile vedere...che non si vede nulla oltre allo stato finale 01010101 ovvero 85. Infatti il PIC a 20MHz è troppo veloce per i nostri occhi. Prima di tutto si osservi che il valore in uscita non è `MAX_VALUE` ma `MAX_VALUE-1` visto che il conteggio partiva da 0 e il numero di volte che è stato eseguito il loop è `MAX_VALUE`.

Per poter vedere il conteggio è necessario rallentare il tutto con un ritardo o delay. Un modo per ottenere questo è per mezzo di un secondo conteggio a vuoto di un ciclo `for`.

Vediamo un altro semplice esempio di ciclo `for` in cui vengono utilizzati anche gli Array in modo da prendere dimestichezza con entrambi e vedere come inserire anche un ritardo. In questo programma sono anche illustrate alcune pratiche di programmazione che è bene non usare, anche se il programma funzionerà comunque senza problemi.

<sup>81</sup> In questo caso si è parlato solo di `i++` e `i--` ma in realtà si possono scrivere anche altre espressioni.

```

#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF         Disabilitato il watchdog timer
//LVP = OFF         Disabilitato programmazione LVP
//PBADEN = OFF      Disabilitato gli ingressi analogici

#define MAX_VALUE 10

void main (void){

    // Variabile per il Conteggio e Delay
    unsigned char i;
    unsigned int j;

    // Definizione dell'Array
    unsigned char mioArray [MAX_VALUE];

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializzo il mio Array
    for (i = 0; i < MAX_VALUE; i++)
        mioArray[i] = i;

    for (i=0; i < MAX_VALUE; i++) {

        // Ritardo
        for (j =0; j < 64000; j++);

        LATD = mioArray[i];
    }

    // Ciclo infinito
    while (1) {

    }
}

```



---

Anche in questo caso non c'è molto da dire per la parte iniziale del programma. Si noti solo che la nostra costante `MAX_VALUE` è stata adesso utilizzata in più punti, dunque eventuali suoi cambi risulteranno molto agevolati, senza dover cambiare più punti del codice sorgente. Il nostro Array è stato inizializzato utilizzando un ciclo `for`, per mezzo del quale è possibile caricare il valore dell'indice all'interno della variabile puntata nell'Array dall'indice stesso.

```
// Inizializzo il mio Array
for (i = 0; i < MAX_VALUE; i++)
    mioArray[i] = i;
```

Tale parte del codice è meglio scriverla in questo modo:

```
// Inizializzo il mio Array
for (i = 0; i < MAX_VALUE; i++) {
    mioArray[i] = i;
}
```

ovvero facendo uso delle parentesi graffe. In questo modo si metterà in evidenza che nel ciclo `for` viene svolta solo questa operazione. Vediamo ora il nostro nuovo loop :

```
for (i=0; i < MAX_VALUE; i++) {

    // Ritardo
    for (j =0; j < 64000; j++);

    LATD = mioArray[i];
}
```

L'inizializzazione del primo loop, ovvero ciclo `for`, è praticamente uguale al precedente. Al suo interno è possibile vedere che sulla `PORTD`, piuttosto che porre il valore dell'indice `i` si pone il valore dell'Array che viene scansionato cella cella per mezzo del loop stesso. In ogni modo ogni cella `i` dell'Array, come sappiamo dalla nostra inizializzazione ha il valore di `i`. La differenza in questo loop, rispetto all'esempio precedente, è dovuto al fatto che è presente al suo interno un secondo loop, il cui scopo è semplicemente contare senza fare nulla, ovvero perdere solo tempo...in questo caso utile per rendere visualizzabile il conteggio. Si noti che il ciclo `for` possiede il punto e virgola alla fine della sua dichiarazione, visto che non possiede nessuna istruzione. Questa linea di codice è in realtà meglio scriverla:

```
for (j =0; j < 64000; j++) {
    // Non faccio nulla
}
```

In questo modo si mette chiaramente in evidenza che il ciclo rappresenta solo un ritardo. Infatti quando si ha un ciclo di questo tipo è facile fraintendere che l'istruzione successiva venga eseguita 64000 volte! Altra pratica da evitare in cicli annidati è l'utilizzo di variabili dal nome `i,j,x,y,z...`. Si immagini una matrice a due dimensioni che venga scansionata `n` volte...e avere indici del tipo `i,x,y`! Quale codice si legge meglio? Questo:

```
for (i = 0; i < 10; i++)
    for (x = 0; x < 100; x++)
        for (y = 0; y < 100; y++) {

            mat[x][y] = (i * 3) + x;
        }
```

---

```
}
```

...o questo secondo codice?

```
for (num_scan = 0; num_scan < NUMERO_SCANSIONI_MAX; num_scan)
    for (riga = 0; riga < MAX_RIGA; riga++)
        for (colonna = 0; colonna < MAX_COLONNA; colonna++) {

            matrice[riga][colonna] = (num_scan * 3) + riga;

        }
}
```

Un'ultima nota sul ciclo `for` riguarda il caso particolare in cui non venga scritta nessuna espressione, come sotto riportato:

```
for ( ; ; ) {
    //ciclo infinito
}
```

Quello che si ottiene è un ciclo infinito, ovvero il PIC eseguirà all'infinito il blocco di istruzioni del ciclo `for`. Il ciclo risulta infinito poiché non c'è nessuna condizione da verificare. Il suo comportamento è praticamente uguale all'istruzione `while (1)`.

---

## Istruzione condizionale if ( )

In ogni programma è di fondamentale importanza poter controllare una variabile o un particolare stato, e decidere se fare o meno una determinata operazione. In C è possibile “porre domande e decidere”, facendo uso dell'istruzione `if (...)` ovvero *se (...)*. Per questa istruzione sono presenti due diverse sintassi, la prima è:

```
if (espressione_logica) {  
    // Programma da eseguire se l'espressione logica è verificata  
}
```

La seconda sintassi è:

```
if (espressione_logica) {  
    // Programma da eseguire se l'espressione logica è verificata  
} else {  
    // Programma da eseguire se l'espressione logica non è verificata  
}
```

Per mezzo della prima sintassi è possibile eseguire una parte di programma se l'espressione logica all'interno delle parentesi tonde è verificata. Per espressione logica si intende una qualunque espressione ottenuta per mezzo degli operatori logici. In particolare un'espressione logica si dice verificata se pari ad un qualunque valore maggiore o uguale a 1, mentre per espressione logica non verificata si intende un valore minore di 1. Le espressioni logiche tradizionali, precedentemente viste, ritornano 1 se l'espressione logica è verificata 0 altrimenti. Come per il ciclo `for (...)` anche per l'istruzione `if (...)` il programma da eseguire è contenuto all'interno di parentesi graffe.

Per mezzo della seconda sintassi, in cui è presente anche la parola chiave `else`, è possibile eseguire un secondo blocco d'istruzioni qualora l'espressione logica non sia verificata. Il secondo blocco viene introdotto per mezzo della parola chiave `else`. Vediamo qualche semplice esempio con la prima sintassi:

```
if ( i == 3 ) {  
    LATD = i;  
}
```

In questo esempio PORTD avrà in uscita il valore di `i` solo se `i` è uguale a 3. Dunque se `i`, nel momento in cui viene effettuato il controllo valesse 2, il blocco d'istruzioni all'interno delle parentesi graffe non verrà eseguito. Si fa notare che per verificare che `i` sia uguale a 3 bisogna scrivere `i==3` e non `i=3` (in cui si assegna 3 ad `i`). Questo tipo di errore è tipico se si ha esperienza di programmazione con il Basic o Pascal. Il C non ci viene in generale in aiuto nella risoluzione di questi problemi, anche se sono presenti compilatori che rilasciano una warning nel caso in cui vedono assegnazioni sospette come espressioni logiche. Il C infatti permette al programmatore di scrivere anche `i=3` come nel seguente esempio:

```
if ( i = 3 ) {  
    LATD = i;
```

---

```
}
```

Il compilatore non segnalerà nessun errore e il programma verrà anche eseguito dal PIC; il problema sta nel fatto che quando viene eseguita l'operazione `if (i=3)` l'espressione logica sarà sempre verificata poiché sarà maggiore di 1, ovvero 3, dunque per il C sarà vera. Dunque come effetto collaterale si avrà che PORTD verrà impostata a 3 ogni volta che verrà eseguito l'`if (i=3)`.

Vediamo ora un esempio completo in cui si effettua la lettura di un pulsante collegato tra massa e RB4 e si pilotano dei LED sulla PORTD.

```
#include <p18f4550.h>
#include <portb.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilitato il watchdog timer
//LVP = OFF          Disabilitato programmazione LVP
//PBADEN = OFF       Disabilitato gli ingressi analogici

void main (void) {

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Abilita i resistori di pull-up sulla PORTB
    EnablePullups();

    // Ciclo infinito
    for (;;) {

        if (PORTBbits.RB4 == 0) {

            // Ho premuto il pulsante su 0 su RB4
            LATD = 0x0F;

        }
        else {

            // Il pulsante è aperto
```

```

        LATD = 0xF0;
    }
}
}

```

Questa volta il programma diventa interessante...comincia ad essere un po' più utile. Come prima cosa si noti che tra i file inclusi è presente:

```
#include <portb.h>
```

Questo file è una libreria fornita da Microchip per mezzo della quale è possibile modificare alcune proprietà della PORTB<sup>82</sup>. La PORTB ha infatti diverse linee di interrupt<sup>83</sup> e dei resistori di pull-up interni. In questo programma dal momento che si vuole leggere un pulsante collegato tra massa e RB4 di Freedom II, si farà uso dei resistori di pull-up. Quando si legge lo stato logico di un pulsante o un interruttore è sempre necessario avere un resistore di pull-up o di pull-down<sup>84</sup>. In questo caso dal momento che la PORTB possiede al suo interno dei resistori di pull-up...perché non sfruttarli?!

Per poter attivare i resistori di pull-up bisogna richiamare la funzione `EnablePullups()`<sup>85</sup> che setta un bit particolare all'interno dei registri del PIC. Per poter leggere il pulsante si è provveduto a realizzare un numero infinito di letture<sup>86</sup> sul pin RB4 di PORTB. Per fare questo si è fatto uso del ciclo `for` senza parametri tanto per non usare il tipico `while (1)`.

All'interno del ciclo infinito viene effettuato il controllo del bit RB4 di PORTB per mezzo della variabile `PORTBbits.RB4`. poiché sono stati attivati i resistori di pull-up e il pulsante è collegato verso massa si ha che normalmente il valore di RB4 è pari a 1 logico, dunque il controllo effettuato dall'`if` vale 0 e viene dunque eseguito il blocco di istruzioni dell'`else`, dunque PORTD viene caricato con il valore 0xF0, ovvero i LED collegati sui quattro bit più significativi saranno accesi, tali LED rimarranno accesi fino a quando non si premerà il pulsante. Quando si premerà il pulsante, il pin RB4 varrà infatti 0 logico (collegato a massa), dunque la condizione `if` verrà verificata e PORTD varrà dunque 0x0F, ovvero si accenderanno i LED sulla PORTD associati ai bit meno significativi. Poiché il ciclo `for` è infinito RB4 verrà continuamente testato, dunque quando si rilascerà il pulsante PORTD varrà nuovamente 0xF0.

Normalmente quando si leggono dei pulsanti la procedura ora utilizzata non è sufficiente. Infatti quando si legge un pulsante è sempre bene accertarsi che il pulsante sia stato effettivamente premuto e in particolare non interpretare una singola pressione come più pressioni. Infatti quando si preme un pulsante si vengono a creare degli spike, ovvero

<sup>82</sup> Microchip fornisce anche altre librerie pronte per l'uso con tanto di sorgente. Per ulteriori informazioni si rimanda alla documentazione ufficiale che è possibile trovare nella cartella doc presente nella cartella d'installazione del C18.

<sup>83</sup> Si ricorda che i 4 bit più significativi del PIC possiedono un interrupt sul cambio di livello logico del pin, quindi viene generato un interrupt sia quando uno di questi ingressi passa da 0 a 1 che quando passa da 1 a 0.

<sup>84</sup> Un resistore di pull-up collega un ingresso a Vcc, mentre un resistore di pull-down collega un ingresso a massa. Questi resistori, o l'uno o l'altro risultano indispensabili quando si deve leggere uno stato di un pulsante o un interruttore. Infatti quando il pulsante/interruttore è aperto l'ingresso rimarrebbe fluttuante ovvero ad un livello logico indeterminato, mentre per mezzo del resistore l'ingresso è vincolato o a Vcc o a GND. L'utilizzo dell'uno o dell'altro dipende da come si collega il pulsante. Se si collega il pulsante a massa, si avrà bisogno di un resistore di pull-up mentre nel caso si colleghi il pulsante a Vcc, si avrà bisogno di un resistore di pull-down.

<sup>85</sup> I resistori di pull-up sono attivati su tutti i bit della PORTB che risultano configurati come ingressi, infatti se un bit è configurato come uscita, tale resistore viene disattivato.

<sup>86</sup> La tecnica di leggere continuamente lo stato di un ingresso per catturarne una sua variazione di stato viene detta polling. Questa si contrappone alla tecnica dell'interrupt (interruzione) in cui il microcontrollore è libero di svolgere altre cose invece di leggere continuamente un ingresso, poiché verrà avvisato (interrupt) quando l'ingresso ha subito una variazione.

l'ingresso del PIC avrà un treno di 0 e 1, che se non opportunamente filtrati possono essere interpretati come pressioni multiple del pulsante stesso.

Per filtrare l'ingresso a cui è collegato il pulsante si inserisce generalmente una pausa dopo aver rilevato la pressione del pulsante stesso e si effettua poi una seconda lettura per essere certi che il pulsante sia stato effettivamente premuto. Questa tecnica può essere implementata sia per mezzo di hardware esterno che per via software, ed è nota come filtro antirimbato. Nel nostro caso effettueremo il filtro per mezzo del software...risparmiando dunque componenti esterni. Nel seguente esempio si incrementa una variabile e la si pone in uscita ad ogni pressione del pulsante posto sul pin RB4:

```
#include <pl8f4550.h>
#include <portb.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

void main (void){

    // Variabile usata per creare un conteggio fittizio di pausa
    int i;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Abilita i resistori di pull-up sulla PORTB
    EnablePullups();

    // Ciclo infinito
    for (;;) {

        if (PORTBbits.RB4 == 0) {

            // Pausa che filtra gli spike
            for (i=0;i<25000; i++) {

            }

        }

    }

}
```

```

        // Controllo nuovamente il pulsante per vedere
        // se è ancora premuto
        if (PORTBbits.RB4 == 0) {

            // Ho premuto il pulsante su 0 su RB4
            LATD++;

        }
    }
}

```

Questo programma almeno all'inizio è molto simile al precedente. Unica differenza è l'introduzione della variabile `i` utilizzata per implementare una pausa per mezzo di un ciclo `for`. Si può notare che all'interno del ciclo infinito il filtro antirimbato viene realizzato nel seguente modo: si effettua prima la lettura del pulsante, qualora questo risulti attivo c'è la piccola pausa del conteggio fittizio<sup>87</sup>. Dopo il filtraggio viene nuovamente riletto il pulsante per vedere se è ancora premuto. Qualora sia ancora premuto vuol dire che effettivamente si è premuto il pulsante. In questo caso quello che viene fatto è incrementare il registro di uscita LATD, per cui si vedrà in uscita il conteggio binario da 0 a 255. Una volta che LATD arriverà a 255 il conteggio riprenderà da 0. Inoltre per come è impostato il programma, tenendo premuto il pulsante il conteggio viene fatto in automatico (come quando si imposta l'ora dell'orologio digitale) e premendo il pulsante troppo rapidamente verrà considerata una sola pressione<sup>88</sup>. Si lascia al lettore la modifica del programma in maniera tale da ottenere un solo incremento anche se si tiene premuto il pulsante.

Qualora il conteggio del ciclo `for` venisse tolto, si può notare come a causa degli spike del pulsante il conteggio verrà fatto quasi come se il microcontrollore stia dando i numeri... L'incremento sarà sempre un bit alla volta ma verranno contati tutti gli spike...creando così apparenti incrementi a passi di 50 e oltre...ma in maniera casuale!

Si fa notare che spesso i pulsanti non vengono letti in polling piuttosto si fa uso delle interruzioni. Questo argomento è però oggetto di un altro paragrafo.

Frequentemente si ha l'esigenza di testare più condizioni al fine di prendere una decisione, in particolare se si devono controllare due condizioni logiche e decidere se accendere un LED, un motore o qualsiasi altro dispositivo, ci si potrebbe porre la domanda: `condizione_1` e `condizione_2` sono entrambe verificate? O ancora, si è verificato solo un evento?

Per rispondere alla prima domanda si potrebbe fare in questo modo, utilizzando due `if` in successione:

```

if (condizione_1 == 1)
    if (condizione_2 == 1){

        // Programma da eseguire se le condizioni sono entrambe verificate
    }

```

Un altro modo che si potrebbe utilizzare sarebbe per mezzo dell'operatore logico *and* `&&`, ovvero:

```

if ((condizione_1 == 1) && (condizione_2 == 1)) {

```

<sup>87</sup> In generale per ottenere un buon filtro antirimbato si raccomandano circa 5-10ms di attesa.

<sup>88</sup> Si capisce che estendendo di molto il filtro antirimbato si può creare la funzione tipica dei pulsanti ON-OFF, ovvero di non accendere o spegnere il sistema se non si tiene premuto il pulsante per almeno 1-2 secondi. In questo modo si filtrano oltre che agli spike, anche le pressioni accidentali.

---

```
// Programma da eseguire se le condizioni sono entrambe verificate
}
```

Si noti che ogni singola espressione è posta all'interno di parentesi tonde. In particolare è possibile utilizzare l'operatore `&&` anche con più termini, ma è bene, per ragioni di leggibilità non eccedere con la fantasia. Oltre all'operatore `&&` si sarebbe potuto utilizzare anche l'operatore *or* `||`, qualora si fosse stati interessati ad eseguire del codice nel caso una od entrambe le condizioni fossero state verificate.

Vediamo qualche dettaglio riguardo all'utilizzo della parola chiave `else` negli esempi appena descritti:

```
if ((condizione_1 == 1) && (condizione_2 == 1)) {
    // Programma da eseguire se le condizioni sono entrambe verificate
} else {
    // Codice eseguito se solo una condizione o nessuna sono verificate.
}
```

In questo caso capire quando viene eseguito il blocco `else` è piuttosto semplice poiché abbiamo a che fare con un solo `if`. Nel nostro caso se l'espressione logica, composta dal verificarsi contemporaneo della condizione\_1 e condizione\_2, non venisse verificata, ovvero una sola condizione o nessuna delle due è verificata, verrebbe eseguito il blocco `else`.

Riprendendo il primo esempio, in cui si è invece fatto utilizzo di due `if`:

```
if (condizione_1 == 1)
    if (condizione_2 == 1){
        // Programma da eseguire se le condizioni sono entrambe verificate
    } else {
        // Programma da eseguire se la condizione 1 è vera e la 2 no!
    }
}
```

Il nostro blocco `else`, apparentemente posto come il precedente ha un ruolo diverso, infatti appartiene al blocco del secondo `if` e non del primo! Per tale ragione le istruzioni del blocco `else` vengono eseguite qualora la condizione\_1 è verificata e la condizione\_2 non è verificata, il che è diverso da quello ottenuto precedentemente! Riscrivendo il codice in questo modo:

```
if (condizione_1 == 1) {
    if (condizione_2 == 1){
        // Programma da eseguire se le condizioni sono entrambe verificate
    } else {
        // Programma da eseguire se la condizione 1 è vera e la 2 no!
    }
} else {
    // Programma da eseguire se la condizione 1 è falsa.
}
```



---

```
} // Lo stato della 2 è indifferente!
```

In questo secondo esempio, è stata inserita la parentesi graffa anche per il primo `if`, creando in questo una chiara evidenza a chi appartiene ogni `else`, l'indentazione aiuta molto alla comprensione. Questo codice si sarebbe potuto scrivere anche in questo secondo modo, ma lo sconsiglio poiché ancor meno leggibile del primo.

```
if (condizione_1 == 1)
    if (condizione_2 == 1) {

        // Programma da eseguire se le condizioni sono entrambe verificate
    } else {

        // Programma da eseguire se la condizione 1 è vera e la 2 no!

else {

    // Programma da eseguire se la condizione 1 è falsa.
    // Lo stato della 2 è indifferente!

}
```

In questo caso il primo `else` che incontriamo è quello precedentemente descritto, mentre il secondo viene ad essere eseguito se la condizione\_1 non è verificata, indipendentemente dalla condizione\_2.

Da quanto detto si capisce che utilizzare due `if` non è proprio come utilizzare l'operatore *and*, se non quando condizione\_1 e condizione\_2 sono entrambe verificate. Dal momento che questo potrebbe essere quello che in realtà interessa, ovvero accendere qualcosa se le due condizioni sono verificate, tale soluzione può essere utilizzata al posto dell'*and*. Un problema degli `if` in successione è legata al fatto che possono rendere il codice poco leggibile per tale ragione, in alcuni casi, piuttosto che concatenare molti `if` è meglio scrivere più blocchi indipendenti.

---

## Istruzione condizionale switch ( )

Quando ci sono molte condizioni da verificare o controllare può ritornare utile, quale alternativa a molti if concatenati, l'istruzione condizionale switch. La sintassi dell'istruzione switch è la seguente:

```
switch (variabile) {  
  
    case valore:  
        // Istruzioni da eseguire  
        break;  
  
    case valore:  
        // Istruzioni da eseguire  
        break;  
  
    case valore:  
        // Istruzioni da eseguire  
        break;  
  
    case valore:  
        // Istruzioni da eseguire  
        break;  
  
    default:  
        // Istruzioni da eseguire  
  
}
```

Dopo la parola chiave `switch` viene posta tra parentesi la variabile di cui si vuole controllare il valore. Tutti i casi da controllare sono posti al fianco della parola chiave `case`. Il valore che si vuole controllare deve essere un valore costante ed intero già noto a priori, ovvero non si può porre come valore un'altra variabile. Questo fatto esclude l'utilizzo dell'istruzione condizionale `case` qualora si vogliano fare confronti tra variabili. In questo caso si deve infatti utilizzare l'istruzione condizionale `if`. Il valore può essere scritto come intero, o come carattere; per esempio:

```
case 128:
```

o anche

```
case 'b':
```

La costante potrebbe anche essere definita per mezzo della direttiva `#define` e si potrebbe utilizzare il suo nome al posto del valore numerico. Questo secondo approccio è in generale quello consigliato in applicazioni professionali. Qualora il confronto tra la variabile e la costante sia verificato, verranno eseguite tutte le istruzioni tra i due punti e la parola chiave `break`. Tale parola chiave dice che dopo le istruzioni il programma deve continuare dopo il blocco d'istruzioni individuato dalle parentesi graffe associate all'istruzione `switch`. Qualora non si facesse uso della parola `break`, ed è lecito ometterla, il programma continuerebbe a controllare le altre condizioni. Si capisce che in effetti ci possono tranquillamente essere applicazioni in cui questo sia effettivamente voluto. Ciononostante, persone con esperienza in BASIC potrebbero omettere l'istruzione `break` per distrazione e non per esigenza. Infatti il linguaggio BASIC non richiede l'istruzione `break`, che in un certo qual modo è eseguita implicitamente. Nel caso in cui nessuno dei `case` viene verificato verrà eseguito il codice che

---

segue la parola chiave `default`. Si noti che in questo caso non è presente la parola chiave `break` poiché non è comunque presente altro codice, ovvero l'esecuzione del programma riprenderebbe comunque dopo le parentesi graffe associate al blocco `switch`. In particolare, qualora non si abbia la necessità di eseguire nessuna operazione di `default` si può omettere il blocco relativo.

Vediamo ora un esempio in cui si leggono i pulsanti della scheda Freedom II e si visualizza il pulsante premuto per mezzo dei LED.

```
#include <pl8f4550.h>
#include <portb.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilitato il watchdog timer
//LVP = OFF          Disabilitato programmazione LVP
//PBADEN = OFF       Disabilitato gli ingressi analogici

#define BT1 0b11100000
#define BT2 0b11010000
#define BT3 0b10110000
#define BT4 0b01110000

void main (void) {

    // Variabile per la lettura pulsanti
    unsigned char button;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Abilita i resistori di pull-up sulla PORTB
    EnablePullups();

    // Ciclo infinito
    while(1) {
```

```

button = PORTB;
button = button & 0xF0;

// Controllo del tasto premuto

switch(button) {

    case BT1:
        LATD = 0x01;
        break;

    case BT2:
        LATD = 0x02;
        break;

    case BT3:
        LATD = 0x04;
        break;

    case BT4:
        LATD = 0x08;
        break;

    default:
        LATD = 0x0F;

}
}
}

```

Il programma oltre alle direttive classiche fa in questo caso utilizzo della direttiva `#define` in modo da definire le costanti che verranno utilizzate all'interno del blocco switch:

```

#define BT1 0b11100000
#define BT2 0b11010000
#define BT3 0b10110000
#define BT4 0b01110000

```

E' possibile vedere che il pulsante BT1 equivale a porre a 0 il bit 5 ovvero RB4. Questo è dovuto al fatto che il pulsante ha un resistore di pull-up ovvero quando è aperto il pin RB4 viene letto come 1, mentre premendo il pulsante, ovvero collegando il pin RB4 a massa RB4 viene letto come 0.

Questa volta la lettura in polling dei pulsanti e il loro controllo è stato posto all'interno del ciclo while (1) che sembrava apparentemente utile solo per bloccare il programma. All'interno del ciclo viene letta la PORTB e vengono azzerati i 4 bit meno significativi:

```

button = PORTB;
button = button & 0xF0;

```

La ragione per cui si azzerano i 4 bit meno significativi discende dal fatto che i pin non utilizzati della PORTB, ovvero i quattro bit meno significativi, sono stati comunque definiti come input. Si ricorda che i pulsanti su Freedom II sono collegati rispettivamente su RB4-RB7. Non conoscendo il valore dei bit RB0-RB3 è bene porli ad un valore noto; avendo definito i valori dei pulsanti BT con i 4 bit meno significativi pari a 0 è dunque necessario porre a 0 i bit meno significativi letti dalla PORTB ovvero della variabile button.

Per fare questo si fa uso dell'operatore *and* &; facendo un *and* con 1 quello che rimane sarà il valore associato al pulsante. Se il pin del pulsante vale 1, fare *and* con 1 vale infatti 1

mentre se il pin valesse 0, facendo *and* 1 il risultato è 0; facendo *and* 0 il bit viene invece posto a 0. Dunque, facendo `button & 0xF0` verranno posti a 0 i bit meno significativi mentre si manterranno al proprio valore i bit relativi ai pulsanti. Qualora si fosse voluto porre ad 1 i 4 bit meno significativi e lasciare invariati i bit più significativi, si sarebbe dovuto fare per mezzo dell'operatore *or* ovvero `|`, ma con il valore `0x0F` e non `0xF0`:

```
button = button & 0x0F;
```

Una volta effettuata la lettura e pulizia della variabile `button`, si effettua il confronto tra la variabile e le varie possibilità prese in considerazione, ovvero la pressione del singolo pulsante. Questo verrà ripetuto in maniera continua per mezzo del ciclo `while` all'interno del quale si è messo l'intero codice. Caricando il codice nella scheda di sviluppo, precedentemente impostata con i Jumper come nel progetto `Hello_World`, è possibile vedere che i LED 0,1,2,3 sono inizialmente accesi, visto che non si è premuto nessun pulsante.

Questo discende dal fatto che `button` vale `11110000`, dunque il confronto con le costanti `BT1-BT4` non viene verificato quindi viene eseguita l'istruzione all'interno del blocco `default`, in cui si accendono i LED 0,1,2,3. Premendo un pulsante alla volta si accenderà il LED relativo, il quale rimarrà acceso fino a quando si terrà premuto il pulsante. Premendo in contemporanea un secondo pulsante si vedrà che verrà eseguito il codice di `default` poiché il programma non riconosce più la pressione del singolo pulsante.

Facciamo ora una digressione. Nel caso in cui si abbia l'esigenza di svolgere un'operazione su di un registro e porre il risultato nel registro stesso è possibile scrivere il codice in altro modo:

```
button &= 0x0F;
```

Quindi ponendo l'operatore prima dell'uguale ed omettendo di riscrivere il nome del registro; questa modalità può essere utilizzata anche per gli altri operatori bitwise e aritmetici. Per esempio si potrebbe scrivere:

```
button |= 0x0F;  
button += 12;  
button -= 3;
```

La ragione per cui si scrive il codice in questo modo è per dare un aiuto al compilatore permettendo di raggiungere un codice ottimizzato. Vediamo come viene eseguita l'operazione *and* nel primo caso ovvero quello standard:

```
53:                                button = button & 0xF0;  
00CA    0EF0    MOVLW 0xf0  
00CC    14DF    ANDWF 0xfdf, W, ACCESS  
00CE    6EDF    MOVWF 0xfdf, ACCESS
```

Effettivamente è quello che ci si aspetta, visto che abbiamo una variabile `unsigned char`, viene infatti eseguita l'operazione *and* tra la costante e `0xF0`. Vediamo ora come viene tradotto il codice aiutando il compilatore ad ottimizzare:

```
55:                                button &= 0xF0;  
00CA    0E00    MOVLW 0  
00CC    90DF    BCF 0xfdf, 0, ACCESS  
00CE    92DF    BCF 0xfdf, 0x1, ACCESS  
00D0    94DF    BCF 0xfdf, 0x2, ACCESS
```

---

```
00D2    96DF    BCF 0xfdf, 0x3, ACCESS
```

...bene, siamo caduti nell'inganno. Volevamo ottimizzare il codice sapendo che per sentito dire potevamo ottimizzarlo con la nuova sintassi. Quello che abbiamo ottenuto è un codice peggiore, ma nel quale è interessante vedere che piuttosto che fare l'*and* vengono posti a zero i bit meno significativi della nostra variabile, che era proprio quello che volevamo fare. Quello che è successo è che molte delle ottimizzazioni in realtà perdono di significato quando si ha a che fare con variabili `char` di un solo byte. Ripetendo l'esempio facendo la stessa operazione ma avendo cambiato la variabile `button` di tipo `unsigned char` in variabile di tipo `int`, si ottiene:

```
53:                                button = button & 0xF0;
0030    CFDE    MOVFF 0xfde, 0xb
0032    F00B    NOP
0034    CFDD    MOVFF 0xfdd, 0xc
0036    F00C    NOP
0038    0EF0    MOVLW 0xf0
003A    160B    ANDWF 0xb, F, ACCESS
003C    0E00    MOVLW 0
003E    160C    ANDWF 0xc, F, ACCESS
0040    C00B    MOVFF 0xb, 0xfde
0042    FFDE    NOP
0044    C00C    MOVFF 0xc, 0xfdd
0046    FFDD    NOP
```

usando la sintassi per ottimizzare il codice:

```
55:                                button &= 0xF0;
00CE    0EF0    MOVLW 0xf0
00D0    16DE    ANDWF 0xfde, F, ACCESS
00D2    6ADD    CLRF 0xfdd, ACCESS
```

In questo caso si ottiene effettivamente un codice ottimizzato. Ancora una volta si mette in evidenza che non bisogna ottimizzare il codice senza verificare i risultati e benefici che si ottengono con l'ottimizzazione. La sintassi `button &= 0x0F`, come anche per gli altri operatori può risultare di più difficile lettura. Dunque se non si hanno ragioni per ottimizzare il codice è meglio preferire la sintassi `button = button & 0x0F` che nel caso di byte ha dato un risultato migliore. Anche se abbiamo avuto un risultato migliore si ricorda che compilatori differenti o anche versioni differenti potrebbero ottimizzare il codice in maniera da avere un'ottimizzazione.

---

## Istruzione condizionale while ( )

L'istruzione `while ( ... )` risulta concettualmente molto simile al ciclo `for ( ... )` ma è più conveniente nei casi in cui non si è a priori a conoscenza del numero di cicli per cui bisogna ripetere un blocco d'istruzioni. Risulta per tale ragione molto utile per il controllo di variabili o registri di stato che possono variare il proprio contenuto in maniera non deterministica. La sintassi dell'istruzione `while ( ... )` ovvero *mentre ( ... )*, è:

```
while (espressione_logica) {  
    // Codice da eseguire fino a quando l'espressione logica è valida  
}
```

Come per l'istruzione `if ( ... )` anche per il ciclo `while`, all'interno delle parentesi tonde è contenuta l'espressione logica che se verificata permette l'esecuzione del gruppo d'istruzioni all'interno delle parentesi graffe. L'espressione logica è verificata se vale 1 o assume un valore maggiore di 1, mentre risulta non verificata se vale 0 o un valore minore di 0.

Qualora l'espressione logica venga verificata, viene eseguito il programma contenuto all'interno delle parentesi graffe. Eseguite le istruzioni contenute tra le parentesi graffe viene eseguito nuovamente il controllo dell'espressione logica. Se l'espressione è nuovamente verificata viene nuovamente eseguito il blocco d'istruzioni tra le parentesi graffe, altrimenti il programma continua con la prima istruzione successiva alle parentesi graffe. Si capisce che se l'espressione logica non fosse verificata al suo primo controllo, il blocco istruzioni del `while` non verrebbe mai eseguito.

Come per il ciclo `for` anche con il ciclo `while` è possibile ottenere dei cicli infiniti se come espressione si scrive qualcosa che viene sempre verificato. Il modo più semplice per ottenere un ciclo infinito si ha semplicemente scrivendo:

```
while (1) {  
    // Istruzioni da eseguire all'infinito  
}
```

Questa istruzione è stata già utilizzata molte volte in maniera più o meno inconsapevole. Un altro modo equivalente potrebbe essere:

```
while (123) {  
    // Istruzioni da eseguire all'infinito  
}
```

Un altro modo potrebbe ancora essere:

```
while (a=5) {  
    // Istruzioni da eseguire all'infinito  
}
```

In questo caso infatti, il risultato dell'espressione è 5 cioè maggiore di 0, dunque equivale ad una condizione sempre “vera”. Si capisce che gli ultimi due modi sono un po' bizzarri anche se effettivamente validi. In particolare l'ultimo esempio potrebbe in realtà essere il risultato di uno sbaglio, ovvero si voleva scrivere l'espressione logica:

```
while (a==5) {

    // Istruzioni da eseguire
}
```

Una volta che si è all'interno di un ciclo `while` vi sono due modi per uscirne. Il primo modo è quello spiegato precedentemente, ovvero l'espressione logica non viene verificata. Questo tipo di uscita non permette però di uscire da un ciclo infinito come quelli precedentemente descritti, o comunque avere delle uscite premature dovute per esempio al verificarsi di errori. In questo caso può risultare comoda l'istruzione `break`. Quando viene eseguita l'istruzione `break` il programma procede con la prima istruzione successiva alle parentesi graffe, dunque esce dal ciclo `while` senza troppe domande. L'istruzione `break` può essere utilizzata anche all'interno di cicli `while` che non siano infiniti, permettendo al programmatore di avere altre condizioni di uscita dal ciclo `while`. Vediamo un semplice esempio di utilizzo di un ciclo `while` in sostituzione del `for`.

```
#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF         Disabilito il watchdog timer
//LVP = OFF         Disabilito programmazione LVP
//PBADEN = OFF      Disabilito gli ingressi analogici

void main (void){

    // Variabile usata per creare un conteggio fittizio di pausa
    unsigned int i;

    // Variabile usata per il conteggio da visualizzare su PORTD
    unsigned char numero;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializzazione della variabile numero
    numero = 0;
```



```

while (numero < 16) {

    // Visualizzo in uscita il valore di numero
    LATD = numero;

    // Incremento della variabile numero
    numero++;

    // Pausa per permettere la visualizzazione del conteggio
    for (i=0;i<64000; i++){

    }

}

// Ciclo infinito
while(1){

}

}

```

La parte iniziale del programma non richiede particolari spiegazioni. Si noti solo che la variabile `i` è stata definita `unsigned int` poiché la pausa è stata fatta effettuando un conteggio pari a 64000, non possibile se la variabile fosse stata `int` (dal momento che un `int` può essere anche un numero negativo). Si noti subito che è possibile osservare tutte le parti funzionali di un'istruzione `for`, ovvero l'inizializzazione della variabile `numero`:

```
numero = 0;
```

Il controllo della variabile `numero`:

```
while (numero < 16)
```

e l'incremento della variabile `numero`:

```
numero++;
```

Questa volta le varie parti sono però nel codice e non parte dell'istruzione `while`, eccetto il test sulla variabile. Si può notare che inizialmente la variabile `numero`, valendo 0 permetterà l'ingresso al blocco istruzioni del ciclo `while`. Una volta dentro vi rimarrà fino a quando la variabile `numero` sarà minore di 16 ovvero fino a 15. Tale valore sarà anche l'ultimo valore visualizzato sui LED ovvero 0x0F, in binario 00001111. Dopo l'incremento è presente il classico ciclo di ritardo che permette un'attesa umana per visualizzare il conteggio. Per testare il programma sulla Scheda Freedom II si impostino i Jumper nella stessa posizione utilizzata per il progetto Hello\_World.

Ora vediamo un altro esempio molto simile al precedente in cui si utilizza l'istruzione `break` per avere un'uscita prematura dal ciclo `while`.

```

#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

```

```

//OSC = HS      Impostato per lavorare ad alta frequenza
//WDT = OFF     Disabilitato il watchdog timer
//LVP = OFF     Disabilitato programmazione LVP
//PBADEN = OFF  Disabilitato gli ingressi analogici

void main (void){

    // Variabile usata per creare un conteggio fittizio di pausa
    unsigned int i;

    // Variabile usata per il conteggio da visualizzare su PORTD
    unsigned char numero;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializzazione della variabile numero
    numero = 0;

    while (numero < 16) {

        // Visualizzo in uscita il valore di numero
        LATD = numero;

        // Incremento della variabile numero
        numero++;

        // Controllo extra sulla variabile numero
        if (numero == 10) {
            break;
        }

        // Pausa per permettere la visualizzazione del conteggio
        for (i=0;i<64000; i++){

        }

    }

    // Ciclo infinito
    while(1){

    }

}

```

---

Il nuovo esempio è praticamente identico al precedente se non per l'aggiunta di un controllo sul valore della variabile `numero` dopo il suo incremento. Il controllo viene effettuato con l'istruzione `if`. In particolare se `numero` è uguale a 10 viene eseguita l'istruzione `break` altrimenti il programma è identico a prima. Questo significa che questa volta il conteggio si fermerà a 10 e non più a 15. Si noti che il conteggio si fermerà a 10 ma `PORTD` varrà 9 ovvero 00001001, infatti raggiunto il valore 10 la variabile `PORTD` non viene più aggiornata.

---

## Le funzioni

Abbiamo fin ora introdotto la funzione `main` quale funzione principale, che viene chiamata all'avvio del nostro programma. Si è compreso che la funzione `main` altro non è che un gruppo d'istruzioni delimitato da parentesi graffe. Qualora tutto il nostro programma dovesse essere inserito all'interno della sola funzione `main`, si renderebbe il codice particolarmente complesso e di difficile lettura. Per mezzo delle funzioni, ogni programma può essere scritto in maniera molto più snella e leggibile e al tempo stesso si ha la possibilità di riutilizzare codice già scritto sotto forma di librerie.

La funzione è un insieme d'istruzioni identificate da un nome e alla quale è possibile passare un certo insieme di variabili ovvero parametri. La funzione può inoltre, a seguito di una elaborazione dati, rilasciare un risultato ovvero avere un valore di ritorno. La sintassi per una funzione è:

```
tipo_di_ritorno nomeFunzione (tipo1 var1, ..., tipoN varN) {  
}
```

Si può subito osservare che la funzione può avere un numero qualunque di parametri in ingresso ma ritorna un solo valore di un determinato tipo. Questo non significa che può ritornare un solo risultato, se infatti si fa uso di una struttura, e si riporta un puntatore a tale struttura (si vedranno maggiori dettagli sulle strutture e puntatori quando si introdurrà in maggior dettaglio il tipo stringa).

Se paragoniamo la sintassi della funzione `main` con quella generale, si può notare che questa non ha nessun risultato di ritorno e non ha nessuna variabile in ingresso<sup>89</sup>.

Vediamo un esempio pratico in cui si abbia l'esigenza di creare una funzione per fare la somma. Si capisce che il valore di ritorno che ci si aspetta deve essere il risultato ovvero la somma, mentre i parametri che sarà necessario passare alla funzione sono gli addendi. Nell'esempio prenderemo in considerazione il caso di una somma tra due interi i quali per loro natura genereranno un risultato che sarà a sua volta un intero.

```
#include <p18f4550.h>  
  
#pragma config FOSC = HS  
#pragma config WDT = OFF  
#pragma config LVP = OFF  
#pragma config PBADEN = OFF  
  
//OSC = HS          Impostato per lavorare ad alta frequenza  
//WDT = OFF         Disabilito il watchdog timer  
//LVP = OFF         Disabilito programmazione LVP  
//PBADEN = OFF      Disabilito gli ingressi analogici  
  
// Funzione per sommare due numeri interi  
  
int sommaInteri (int add1, int add2) {  
    int somma;
```

---

<sup>89</sup> Questo non è vero in ANSI C, dove la funzione `main` può ricevere variabili in ingresso e ritornare un valore. Le variabili in ingresso sono rappresentate dal testo che si scrive sulla linea di comando quando si lancia il programma stesso da una shell di comando. Il valore di uscita è in generale un valore che segnala un eventuale codice di errore.

```

    somma = add1+add2;

    // Ritorno la somma
    return(somma);
}

void main (void){

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Effettuo la somma tra 3 e 7
    LATD = sommaInteri (3,7);

    // Ciclo infinito
    while(1){

    }

}

```

Come di consueto la parte iniziale del programma relativo alle direttive non è cambiato, ciononostante è possibile vedere che prima della funzione `main` è stata dichiarata la funzione `sommaInteri`.

```

int sommaInteri (int add1, int add2) {

    int somma;

    somma = add1+add2;

    // Ritorno la somma
    return(somma);
}

```

Questa potrebbe anche essere riscritta:

```

int sommaInteri (int add1, int add2) {

    // Ritorno la somma
    return(add1+add2);
}

```

---

In questo modo si risparmia anche una variabile. Il fatto che la funzione `sommaInteri` sia definita prima della funzione `main` non è obbligatorio, ma si ritornerà a breve su questo argomento. Vediamo qualche dettaglio sulla definizione della funzione:

```
int sommaInteri (int add1, int add2)
```

E' possibile vedere che si è dichiarato, come tipo di ritorno, un `int`. Questo discende dal fatto che la somma tra due interi è sempre un intero<sup>90</sup> siano essi positivi o negativi. I due addendi vengono dichiarati all'interno delle parentesi tonde. Questa dichiarazione è a tutti gli effetti una dichiarazione di variabile ma come si vedrà nel paragrafo sulla visibilità delle variabili queste variabili verranno rilasciate non appena la funzione avrà termine. Le variabili dichiarate tra parentesi ospiteranno i valori degli addendi che verranno inseriti nel momento della chiamata della funzione stessa. Il corpo della funzione è praticamente identico alla funzione `main`, ovvero si possono dichiarare altre variabili e si può scrivere tutto il codice che si vuole. Una differenza è la presenza dell'istruzione `return ( )` (in realtà è una funzione) che permette di ritornare il risultato della somma. Qualora la funzione non abbia nessun valore di ritorno ovvero il valore di ritorno sia `void`, come potrebbe essere per una funzione di ritardo, la funzione `return ( )` non risulta necessaria. In particolare se si dichiarasse il valore di ritorno di tipo `void` e si facesse comunque uso della funzione `return ( )` verrebbe generato sia un errore che una warning:

*Warning [2052] unexpected return value*  
*Error [1131] type mismatch in assignment*

All'interno delle parentesi tonde della funzione `return ( )` si deve porre una variabile o costante che sia dello stesso tipo di quella che è stata dichiarata per il ritorno della funzione. In particolare il valore ritornato sarà il risultato della nostra funzione.

Dopo un'istruzione `return ( )` si ha l'uscita dalla funzione e il programma riprende dal punto in cui la funzione era stata chiamata. All'interno di ogni funzione possono essere presenti anche più punti di uscita, ovvero `return ( )` ma solo uno sarà quello che effettivamente farà uscire dalla funzione stessa. Se la funzione non ha nessun valore di ritorno, si ha l'uscita dalla funzione quando il programma giunge alle parentesi graffe. Quando questo avviene il programma riprende dall'istruzione successiva alla chiamata della funzione. Da quanto detto riguardo lo Stack Pointer e lo Stack Memory si capisce che la chiamata ad una funzione equivale a posizionare il Program Counter al punto della memoria in cui è definita la funzione. Lo Stack Pointer verrà incrementato e l'indirizzo di ritorno verrà memorizzato all'interno della Stack Memory. Questo significa che all'interno di una funzione è possibile richiamare altre funzioni ma bisogna tenere sempre a mente del limite imposto dalla Stack Memory disponibile<sup>91</sup>.

La chiamata alla funzione avviene nel seguente modo:

```
LATD = sommaInteri (3,7);
```

Si noti che nel caso specifico si sono passati come parametri due costanti, ma si sarebbero potute passare anche due variabili di tipo intero<sup>92</sup>. Dal momento che la funzione ritorna un

---

<sup>90</sup> In questo esempio non si sta considerando il caso in cui il risultato possa eccedere il valore massimo consentito da un intero.

<sup>91</sup> Si faccia riferimento al Capitolo relativo all'architettura dei PIC18 per maggiori informazioni.

<sup>92</sup> Si fa presente che le variabili in C sono passate per valore e non per riferimento. Questo significa che cambiando il valore della variabile passata per valore, il cambiamento non si ripercuote sulla variabile originale. Il cambiamento avrà influenza solo all'interno della funzione stessa. Per passare una variabile per riferimento bisogna utilizzare i puntatori che verranno descritti nei prossimi Capitoli.

risultato bisogna scrivere sulla sinistra della funzione un'assegnazione, o comunque la funzione deve essere parte di una espressione il cui risultato verrà assegnato ad una variabile; se così non si facesse il risultato verrebbe perso. Nel nostro caso il valore della somma, pari a 10 (00001010) viene posto in uscita alla PORTD<sup>93</sup>. Il valore di ritorno viene spesso usato anche per ritornare un codice di errore in maniera da avvisare chi chiama la funzione, del fatto che si è verificato un errore. Per convenzione un valore di ritorno negativo o nullo equivale alla presenza di un errore. Un valore di ritorno pari ad 1 o maggiore di 0 equivale ad una corretta esecuzione delle operazioni contenute nella funzione. In questi casi non si ha sempre interesse a gestire gli errori, anche se è buona abitudine farlo, perciò il valore di ritorno può non essere assegnato a nessuna variabile.

Come detto ogni funzione deve essere dichiarata prima della funzione main ma questo non è obbligatorio. Se si dovesse dichiarare la funzione o funzioni dopo la funzione main il compilatore darà un avviso poiché compilando la funzione main si troverà ad usare la funzione `sommaInteri` che non è stata precedentemente dichiarata...ma che è presente dopo la funzione main. In particolare se si ripete il programma precedente semplicemente spostando la funzione `sommaInteri` dopo la funzione main, si avrà il seguente messaggio di warning:

*Warning [2058] call of function without prototype*

Il programma verrà in ogni modo compilato senza problemi, ma è bene sempre compilare il programma senza alcun messaggio di warning. Per evitare questo messaggio, prima della funzione main bisogna scrivere il prototipo di funzione ovvero una riga in cui si avvisa il compilatore che la funzione `sommaInteri` che è utilizzata all'interno della funzione main è dichiarata dopo la funzione main stessa. La dichiarazione di un prototipo di funzione si effettua semplicemente per mezzo del nome della funzione stessa e il tipo di variabili presenti., terminando il tutto per mezzo di un punto e virgola, ovvero senza il corpo del programma associato alla funzione. Il prototipo di funzione e relativo spostamento della funzione `sommaInteri` è riportato nel seguente esempio:

```
#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS      Impostato per lavorare ad alta frequenza
//WDT = OFF     Disabilitato il watchdog timer
//LVP = OFF     Disabilitato programmazione LVP
//PBADEN = OFF  Disabilitato gli ingressi analogici

// Prototipo di funzione
int sommaInteri (int add1, int add2);

void main (void) {

    // Imposto PORTA tutti ingressi
```

<sup>93</sup> Per simulare il progetto sulla scheda Freedom II si impostino i Jumper prima della programmazione, nella stessa posizione utilizzata nel progetto Hello\_World.

```

LATA = 0x00;
TRISA = 0xFF;

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

// Effettuo la somma tra 3 e 7
LATD = sommaInteri (3,7);

// Ciclo infinito
while(1) {
}

}

// Funzione per sommare due numeri interi
int sommaInteri (int add1, int add2) {

    int somma;

    somma = add1+add2;

    // Ritorno la somma
    return (somma);
}

```

Il fatto di dichiarare il prototipo di funzione permette al compilatore di effettuare il controllo del tipo di variabili che vengono passate alla funzione stessa, dunque è cosa buona sfruttare questo vantaggio eliminando il messaggio di warning per mezzo della dichiarazione del prototipo di funzione.

L'utilizzo delle funzioni è uno strumento molto potente che permette di risolvere applicazioni complesse concentrandosi su singoli problemi, inoltre, come detto, risulta anche un modo per riutilizzare il codice<sup>94</sup>. Per fare questo si può semplicemente copiare ed incollare una funzione da un programma in un altro o cosa migliore creare una propria libreria che contenga una certa categoria di funzioni. Si può per esempio fare una libreria per controllare un LCD o una libreria per controllare la comunicazione con un integrato particolare. Per poter scrivere una libreria basta creare un altro file, come si è fatto per il file main.c e dargli un nome per esempio funzioniLCD.c ed aggiungerlo al progetto. Il file creato in questo modo è

<sup>94</sup> Devo ammettere che una volta che imparate un linguaggio come C++ e Java, il riciclare il codice per mezzo di funzioni rappresenta un metodo piuttosto scomodo ed inefficace. I linguaggi Object Oriented hanno infatti tecniche più eleganti per il riutilizzo del codice.



---

in realtà parte del progetto stesso e non una vera e propria libreria. Per trattarla come una libreria basta in realtà cancellare il file dal nostro progetto ed includerlo con la direttiva `#include`. Quando si include un file con la direttiva `#include` si hanno due formati, il primo è quello visto fino ad adesso, ovvero `#include <nome_file>` dove si scrive il file da includere tra i due simboli di minore e maggiore. Questo modo è valido solo se il file di libreria è insieme ai file di libreria della Microchip<sup>95</sup>.

Se il file è contenuto altrove bisogna utilizzare quest'altro formato `#include "percorso_file/nome_libreria"` o meglio includere il percorso base `percorso_file` tra i percorsi dove il compilatore cercherà i file. Nel caso particolare in cui la libreria risieda all'interno della directory del nostro progetto basta scrivere `#include "nome_file"`, infatti il percorso della directory madre è automaticamente incluso tra quelli di ricerca. Anche l'includere il file per mezzo della direttiva `#include` non fa in realtà del file una vera e propria libreria. La creazione di una libreria verrà descritta nel prossimo Capitolo.

L'utilizzo di file multipli non è solo consigliato per raccogliere una certa classe di funzioni ma anche per spezzare il programma principale. Infatti, in programmi complessi si potrebbe creare un file che contenga solo la dichiarazione delle variabili ed includerlo come precedentemente detto. Inoltre è buona abitudine scrivere le funzioni in altri file seguendo un criterio di appartenenza e scrivere sul file principale solo lo scheletro del programma. Per un robot si potrebbe scrivere per esempio un file per le sole variabili, un file con le funzioni di controllo del movimento, un file per la gestione degli occhi, un file per il controllo delle interfacce grafiche e via scorrendo.

Quando si fa utilizzo di file multipli bisogna sempre tenere conto della dichiarazione del prototipo di funzione, in maniera da evitare i messaggi di warning.

---

<sup>95</sup> Tra le opzioni di progetto si è settato questo percorso.

---

## Visibilità delle variabili

Introdotte le funzioni, il nostro programma può essere frammentato in molti file o blocchi. A seconda di dove si dichiarerà una variabile questa potrà o meno essere utilizzata, ovvero visibile, in determinate parti del programma. Per visibilità o scope di variabili si intende le parti di programma dove è possibile utilizzare una variabile precedentemente dichiarata.

Una variabile viene detta globale quando può essere utilizzata in qualunque parte del programma. Generalmente questo tipo di variabili vengono dichiarate nel caso in cui l'informazione in esse contenute debba essere condivisa da più moduli ovvero funzioni di programma<sup>96</sup>. Per dichiarare una variabile globale bisogna effettuare una dichiarazione al di fuori della funzione main come nel seguente esempio:

```
// Variabile globale
int temperatura_sistema = 0;

void main (void) {

    // Programma da eseguire

    if (temperatura_sistema > 37) {

        attivaAllarme ();

    }

}
```

Quando una variabile è globale vuol dire che nel PIC gli verrà assegnata la quantità di memoria RAM necessaria per contenerla e questa rimarrà fissa e accessibile da ogni parte del programma.

Quando una variabile viene dichiarata all'interno della funzione main questa sarà accessibile solo da parti di programma interne alla funzione main stessa ma non da funzioni esterne. Ad una variabile interna alla funzione main, come per le variabili globali gli viene assegnata della RAM e questa rimane fissa durante tutto il tempo di esecuzione del programma da parte del PIC ma sarà accessibile solo dalla funzione main.

Quando una variabile è dichiarata all'interno di una funzione generica, come per esempio la variabile `somma` dichiarata all'interno della funzione `sommaInteri` degli esempi precedenti, questa sarà visibile solo all'interno della funzione stessa. In particolare la RAM che viene allocata per la variabile viene rilasciata (resa disponibile) una volta che la funzione termina le sue operazioni; questo significa che la variabile cessa di “esistere”. Accedere ad una variabile locale di una funzione che è stata rilasciata non genera, a livello di compilazione, alcun errore o warning. Ciononostante l'esecuzione di un programma in cui si fa uso di dati contenuti in variabili rilasciate può portare effetti indesiderati e comportamenti imprevedibili. Questo è dovuto al fatto che accedendo ad una variabile rilasciata, il valore che si ottiene può in realtà appartenere ad un'altra variabile temporanea<sup>97</sup>.

Poiché una variabile dichiarata all'interno della funzione main non sarà visibile all'interno di altre funzioni e viceversa, è possibile utilizzare anche nomi di variabili utilizzati in altre funzioni.

---

<sup>96</sup> In linguaggi di programmazione ad oggetti il loro utilizzo è sconsigliato in quanto crea delle relazioni di dipendenza tra classi.

<sup>97</sup> Alcune volte pur accedendo al contenuto di una variabile rilasciata il valore letto è quello esatto, ma questo è solo dovuto a fortuna. Si capisce che non è bene affidare il funzionamento del programma alla fortuna...!

---

Nonostante sia possibile avere nomi di variabili uguali, grazie al fatto che le funzioni non riescono a vedere le variabili interne ad altre funzioni questa non è una buona pratica di programmazione visto che può creare facilmente confusione. Questa possibilità dovrebbe essere sfruttata solo per variabili molto semplici come le variabili di indice utilizzate nei cicli `for` o `while` ma è bene sempre non eccedere.

Alcune volte si può avere l'esigenza di contare degli eventi ogni volta che si richiama una funzione, per esempio il numero di volte che la funzione viene chiamata. Ma come è possibile contare degli eventi se le variabili interne ad una funzione vengono poi cancellate. Il modo per fare questo è dichiarare una variabile globale e utilizzarla per il conteggio interno alla funzione. Infatti tale variabile non appartenendo alla funzione non verrà cancellata ed ad ogni accesso avrà sempre il valore relativo all'ultimo incremento. Un altro modo per raggiungere lo scopo è dichiarare la variabile `static`, ovvero statica. Questo significa che la variabile pur appartenendo alla funzione non verrà cancellata alla fine della funzione stessa. Dunque ogni volta che la funzione verrà rieseguita questa potrà accedere sempre alla stessa variabile precedentemente dichiarata `static`. In particolare una variabile dichiarata `static` viene inizializzata una sola volta nel momento della sua dichiarazione, quindi successive chiamate alla funzione non inizializzeranno nuovamente la variabile. Vediamo il seguente esempio, in cui la funzione `Conteggio` grazie alla variabile statica conta il numero di chiamate effettuate alla funzione stessa.

```
#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS           Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

// Funzione che conta le sue chiamate

int Conteggio (void) {

    // Dichiarazione variabile statica
    static int contatore=0;

    contatore++;

    return (contatore);
}

void main (void){

    // Variabile per il ciclo for
    unsigned char i;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
```

```

LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

for (i = 0; i < 10; i++) {

    Conteggio ();
}

LATD = Conteggio ();

// Ciclo infinito
while(1) {

}
}

```

La dichiarazione della funzione viene fatta come nel caso precedente prima della funzione `main`. Si può osservare che nel nostro caso non viene passato nessun valore, dunque come per la funzione `main` lì si sono dichiarati `void`.

```

int Conteggio (void) {

    // Dichiarazione variabile statica
    static int contatore=0;

    contatore++;

    return (contatore);
}

```

La variabile per il conteggio delle chiamate della funzione stessa viene dichiarata `static` semplicemente ponendo la parola chiave `static` prima della dichiarazione stessa.

```

static int contatore=0;

```

In questo caso l'inizializzazione a zero è stata fatta direttamente con la dichiarazione della variabile stessa. Qualora la variabile non fosse `static` questa verrebbe inizializzata ad ogni chiamata della funzione. Dal momento che la variabile è di tipo `static` viene invece inizializzata a zero una sola volta alla prima chiamata della funzione stessa.

Il resto della funzione non fa altro che incrementare il valore del contatore e ritornarlo come valore di ritorno. Nell'esempio la funzione `Conteggio` viene chiamata 10 volte per mezzo di un ciclo `for`:

```

for (i = 0; i < 10; i++) {

```

---

```
    Conteggio ();  
}
```

Dopo il ciclo for il valore di ritorno della funzione Conteggio viene scritta su PORTD.

```
LATD = Conteggio ();
```

Dal momento che per scrivere il valore di ritorno si è chiamata nuovamente la funzione si ha che il valore finale è 11 ovvero 00001011.

Può essere interessante rimuovere ora la parola chiave static e vedere che in uscita si avrà il valore 00000001. La ragione è legata al fatto che la variabile conteggio viene adesso inizializzata sempre a zero.

Si fa presente, anche se un po' prematuro parlarne, che qualora si ritorni un puntatore di una struttura interna ad una funzione, questa deve essere dichiarata statica, altrimenti la struttura verrà rilasciata ed il puntatore non sarà più valido.

---

## Le interruzioni

Da quanto fin ora spiegato si è potuto capire che i PIC hanno la possibilità di essere utilizzati in sistemi molto complessi in cui sia richiesto di svolgere molte operazioni in contemporanea. In realtà qualunque microcontrollore, per quanto veloce sia, svolgerà un'operazione alla volta; le sue velocità di esecuzione sono tali però da dare l'impressione che svolga molte operazioni in contemporanea. Ogni qual volta si abbia l'esigenza di svolgere più operazioni è bene strutturare il programma in modo da poter gestire il tutto in maniera snella e funzionale. La tecnica del polling potrebbe creare qualche problema poiché si perde molto tempo a controllare periferiche che magari sono inattive. Un esempio potrebbe essere il controllo dei pulsanti, come si è visto negli esempi precedenti. La gestione di più funzioni o processi è notevolmente migliorata per mezzo dell'utilizzo degli interrupt, ovvero segnali inviati dall'hardware quando viene a verificarsi un determinato evento. I PIC possiedono, per ogni hardware interno varie modalità e tipologie d'interrupt in maniera da permettere l'utilizzo dell'hardware non solo per mezzo della tecnica del polling ma anche per mezzo degli interrupt. Per esempio la PORTB per i pin RB4-RB7 genera un evento d'interrupt qualora ci sia il cambio del valore logico sul pin, che potrebbe essere causato per esempio dalla pressione di un pulsante. Consapevoli ora che l'hardware può generare degli interrupt cerchiamo di capire cosa sono e come possono essere sfruttati tali segnali<sup>98</sup>.

### *Che cose' un'interruzione?*

Come dice la parola stessa un'interruzione è un evento che interrompe la normale esecuzione di un programma. Gli eventi che possono interrompere la normale esecuzione di un programma sono molteplici e possono differire da microcontrollore a microcontrollore, ciononostante un'interruzione comune a tutti i microcontrollori è quella che si viene a generare con il segnale di Reset. Infatti il Reset rappresenta un'interruzione che in particolare interrompe la normale esecuzione del programma facendolo iniziare nuovamente da capo. Altre interruzioni tipiche sono quelle che vengono generate da periferiche interne, come per esempio il convertitore analogico digitale (ADC), l'USART, i Timer, le linee sulla PORTB e altro ancora.

Questi tipi d'interruzione, a differenza dell'interruzione generata dal Reset non fanno iniziare il programma da capo ma lo fanno continuare a partire da un punto specifico del programma stesso; questo punto viene detto vettore d'interruzione. Quando avviene un'interruzione da parte delle periferiche, prima di saltare al vettore d'interruzione, l'Hardware<sup>99</sup> si preoccupa di salvare tutte le informazioni necessarie per poter riprendere dal punto in cui il programma è stato interrotto. Come visto in precedenza il PC viene salvato nello Stack in maniera da riprendere il programma dal punto in cui è stato interrotto<sup>100</sup>.

Il PIC18F4550 come ogni PIC18, possiede due livelli d'interruzione, ovvero due vettori di interruzione. Il vettore d'interruzione ad alta priorità posizionato all'indirizzo di memoria 0x08 e il vettore d'interruzione a bassa priorità posizionato all'indirizzo di memoria 0x18.

Ogni interruzione a bassa priorità può interrompere il normale flusso del programma permettendo di svolgere la particolare funzione associata con la gestione delle stesse. Le istruzioni ad alta priorità possono interrompere sia il normale flusso del programma sia

---

<sup>98</sup> Nonostante l'importanza delle interruzioni, in questo paragrafo, non si tratterà in maniera esaustiva l'intero argomento, dal momento che questo può variare da microcontrollore a microcontrollore. Ciò nonostante si tratteranno gli aspetti principali e grazie agli esempi si darà modo al lettore di acquisire gli strumenti necessari per affrontare gli aspetti non trattati. In particolare si rimanda alla documentazione ufficiale del PIC utilizzato per una più approfondita trattazione.

<sup>99</sup> In realtà anche il software interviene spesso per il salvataggio di variabili particolari che altrimenti non verrebbero salvate.

<sup>100</sup> L'eventuale istruzione caricata nell'ISR viene comunque terminata.

---

l'esecuzione della routine associata alle interruzioni a bassa priorità. Da questa possibilità si capisce per quale ragione questa seconda interruzione viene detta ad alta priorità. Lo svolgimento della funzione associata alle interruzioni ad alta priorità non può essere invece interrotta né da un'interruzione ad alta priorità né da un'interruzione a bassa priorità.

Ciononostante, nel caso si dovesse premere il tasto di Reset anche la gestione di un'interruzione ad alta priorità verrebbe interrotta per far iniziare il programma nuovamente da capo; il Reset è in un certo qual modo ad "altissima priorità". Il verificarsi di un interrupt, come detto, causa, qualora sia accettato, il salvataggio del registro PC e il suo cambio in modo da puntare l'Interrupt Vector di competenza. Oltre a questa operazione vengono in realtà salvati i registri WREG, BSR e lo STATUS register. La modalità di salvataggio degli stessi differisce a seconda del tipo d'interruzione. Si capisce intanto che il posto dove vengono salvati deve essere differente per i due tipi d'Interrupt, perché se così non fosse l'Interrupt ad alta priorità potrebbe causare la perdita di dati associati ad un'interruzione a bassa priorità.

Nel caso d'interruzione a bassa priorità i registri ora citati vengono salvati nello Stack per mezzo di normali operazioni MOVFF, MOVWF e MOVF. Nel caso in cui si verifichi un'interruzione ad alta priorità i registri WREG, BSR, STATUS vengono salvati in automatico nei registri *shadow*, ovvero ombra, per mezzo di un solo ciclo di clock. Da quanto detto si capisce che il tempo richiesto per iniziare lo svolgimento della routine associata ad un interrupt è diverso nel caso in cui si tratti di un'interruzione a bassa o alta priorità, in particolare quelle ad alta priorità sono gestite più velocemente. Tale intervallo di tempo è noto come latency, ed è un parametro di fondamentale importanza nel caso in cui si sia realizzando un sistema real time<sup>101</sup>.

Per la gestione delle interruzioni è necessario impostare i seguenti registri:

- RCON
- INTCON
- INTCON2
- INTCON3
- PIR1, PIR2
- PIE1, PIE2
- IPR1, IPR2

In realtà a seconda delle periferiche che sarà necessario gestire solo alcuni registri dovranno essere propriamente impostati, in particolare solo alcuni bit. Ciononostante il registro RCON ed INTCON hanno al loro interno i bit fondamentali per la gestione delle interruzioni. All'interno di tali registri sono presenti tutti i bit ovvero le impostazioni necessarie per la gestione delle interruzioni<sup>102</sup>. Per maggiori informazioni sui singoli registri si rimanda al datasheet del PIC utilizzato.

Ogni tipo d'interruzione che può essere generata da una periferica, possiede tre bit di controllo posizionati in punti diversi nei registri ora citati.

In particolare si ha un bit che funziona da flag per l'interruzione, ovvero quando vale 1 segnala che si è verificata l'interruzione da parte della periferica. Un secondo bit è dedicato all'Enable, ovvero abilitazione, dell'interruzione di una determinata interruzione, mentre il terzo bit serve per decidere se l'interruzione da parte della periferica deve essere considerata ad alta priorità o a bassa priorità.

---

<sup>101</sup> Si ricorda che un sistema real time non è un sistema necessariamente veloce, ma un sistema che garantisce lo svolgimento di determinate operazioni o compiti in un tempo predefinito.

<sup>102</sup> Il modulo USB è un po' un mondo a parte per quanto riguarda le interruzioni. La sua gestione esula dallo scopo di tale testo, per cui si rimanda al datasheet per ulteriori informazioni.

---

Oltre a questi tre bit associati ad ogni tipologia d'interrupt, sono presenti anche altri bit per la gestione globale degli interrupt. I bit per la gestione globale delle interruzioni sono GIE e PEIE del registro INTCON e il bit IPEN del registro RCON. I PIC18 pur potendo gestire le interruzioni ad alta e bassa priorità possono anche lavorare in modalità compatibile con i PIC16. Per lavorare in modalità compatibile bisogna impostare a 0 il bit IPEN, questo è per altro il valore di default. Quando IPEN vale zero, il bit GIE permette di abilitare in maniera globale le interruzioni mentre PEIE permette di abilitare/disabilitare le interruzioni delle periferiche. Quando i bit valgono 1 la funzione è abilitata, mentre quando vale 0 è disabilitata. Quando il PIC18 lavora in maniera compatibile con i PIC16 l'interrupt Vector è posto all'indirizzo 0x08, cioè come se fosse abilitata l'interruzione ad alta priorità. Per lavorare in maniera compatibile le seguenti impostazioni sono richieste:

```
// Abilito modalità compatibile (di default vale già 0)
RCONbits.IPEN = 0;

// Abilito l'interrupt globale
INTCONbits.GIE = 1;

// Abilito interrupt per periferiche
INTCONbits.PEIE = 1 ;
```

In questa configurazione i bit per selezionare la priorità non devono essere impostati, poiché di default valgono 1, ovvero abilitati per funzionare ad alta priorità.

Nel caso si voglia fare uso delle funzionalità avanzate d'interrupt offerte dall'architettura PIC18, bisogna settare i bit precedenti in maniera differente. In particolare è necessario porre ad 1 il bit IPEN. Quando questo bit viene abilitato i bit IPEN e GIE del registro INTCON assumono un altro significato. Il bit GIE, anche nominato GIEH, permette, se posto ad 1, di abilitare le interruzioni ad alta priorità. Il bit PEIE, anche nominato GIEL, se posto ad 1, abilita tutte interruzioni a bassa priorità. In questo caso si hanno i due vettori d'interruzione posti all'indirizzo 0x08 (alta priorità) e 0x18 (bassa priorità). Quando si lavora con i due livelli di priorità è necessario impostare i bit dei registri di priorità IPR e IPR2. Per abilitare le interruzioni con i due livelli di priorità si deve procedere come segue:

```
// Abilito modalità interruzione a due livelli alta e bassa
RCONbits.IPEN = 1;

// Abilito gli interrupt ad alta priorità
INTCONbits.GIEH = 1;

// Abilito gli interrupt a bassa priorità
INTCONbits.GIEL = 1 ;
```

Prima di abilitare gli interrupt, è consigliabile aver precedentemente abilitato ed impostato tutte le periferiche che dovranno generare gli interrupt. Questo è valido anche nella modalità compatibile PIC16.

Per poter gestire le interruzioni è necessario dichiarare una funzione particolare o meglio bisogna specificare al compilatore dove si trova tale funzione. Infatti nel momento in cui il programma viene interrotto, questo andrà alla locazione di memoria 0x08 o 0x18 a seconda del tipo d'interruzione o modalità abilitata. A partire da questi indirizzi non è possibile scrivere l'intero programma di gestione delle interruzioni (si faccia riferimento al Capitolo sull'architettura dei PIC18), si pensi ad esempio che tra il vettore ad alta priorità e bassa priorità sono presenti solo 16 locazioni di memoria. Per tale ragione quello che si fa è



---

posizionare dei salti, che dai vettori d'interruzione posizionano il programma (ovvero il Program Counter) alle relative funzioni di gestione dell'interruzione. Quanto appena detto viene fatto dal seguente segmento di codice:

```
.
.    // Intestazione del programma
.

// Prototipo di funzione
void High_Int_Event (void);

#pragma code high_vector = 0x08

void high_interrupt (void) {

    // Salto per la gestione dell'interrupt
    __asm GOTO High_Int_Event __endasm
}

#pragma code

#pragma interrupt High_Int_Event

void High_Int_Event (void) {

    // Programma per la gestione dell'interruzione
}

void main (void) {

.    // Programma principale
.
}
}
```

Da quanto appena scritto si capisce che in C18 la gestione delle interruzioni è un po' infelice, in particolare si fa uso della direttiva pragma che non è ANSI C. Questo significa che se si volesse compilare il programma con un altro compilatore diverso da C18 bisognerà molto probabilmente modificare la dichiarazione della funzione per la gestione delle interruzioni. Nonostante la nostra infelicità la gestione delle interruzioni è strutturata in maniera sensata e permette tutta la flessibilità che può essere richiesta in fase di progettazione di una nuova applicazione.

Nel segmento di codice sopra scritto si è fatto riferimento all'interruzione ad alta priorità ovvero modalità compatibile ma per quelle a bassa priorità vale la stessa dichiarazione a patto di cambiare il vettore d'interruzione 0x08 con 0x18 e la parola High con Low in modo da chiamare le funzioni in maniera differenti. Il nome delle funzioni può essere scelto dal programmatore, quello riportato è il nome di cui faccio uso per comodità. Altra modifica richiesta per lavorare con le interruzioni a bassa priorità sarà di seguito descritta.

Vediamo i passi che si sono seguiti in questo segmento di codice. Come prima cosa si è dichiarato il prototipo di funzione per la gestione dell'interrupt, visto che tale funzione verrà richiamata prima della sua dichiarazione.

```
void High_Int_Event (void);
```

Come secondo passo si è fatto uso della direttiva `#pragma code` per dichiarare il vettore

---

d'interruzione a cui si sta facendo riferimento.

```
#pragma code high_vector = 0x08
```

La direttiva `#pragma` ha varie funzioni, nel caso specifico è utilizzata in combinazione con la parola `code`, ovvero codice. Questo significa che viene dichiarata una sezione di codice nominata `high_vector` (ma potrebbe essere chiamata diversamente) che partirà dall'indirizzo `0x08`, che guarda caso è proprio il nostro interrupt vector ad alta priorità, ovvero il punto in cui il programma andrà ad eseguire le istruzioni al verificarsi di un'interruzione ad alta priorità. Il terzo passo è la dichiarazione della funzione `high_interrupt` che grazie al passo precedente è posizionata proprio sul vettore d'interruzione di nostro interesse.

```
void high_interrupt (void)
```

Tale funzione deve avere obbligatoriamente sia i parametri d'ingresso che di ritorno di tipo `void`. In questa funzione si inserisce il salto alla funzione vera e propria che gestirà l'interruzione. Tale salto viene effettuato con l'istruzione assembler `GOTO`.

```
_asm GOTO High_Int_Event _endasm
```

Per poter scrivere segmenti di codice assembler all'interno del programma principale è necessario inserire tale codice tra le due parole chiave `_asm` e `_endasm`. Queste parole chiave non sono standard ANSI C, differiscono infatti da compilatore a compilatore ma sono in generale presenti in ogni compilatore.

`_asm` e `_endasm` risultano particolarmente utili per quelle parti di codice che devono essere ottimizzate ma richiedono una conoscenza del codice assembler e soprattutto una buona conoscenza del PIC che si sta utilizzando<sup>103</sup>. Come quarto passo si fa nuovamente uso della direttiva `pragma`;

```
#pragma code
```

In questo caso la direttiva permette di ritornare alla normale gestione del programma, lasciando al compilatore il compito di decidere dove porre il programma. Il quinto passo è ancora una volta l'utilizzo della direttiva `#pragma`, questa volta associata però con la parola chiave `interrupt`:

```
#pragma interrupt High_Int_Event
```

La direttiva `#pragma interrupt` associa alla sessione chiamata in questo caso `High_Int_Event`, la funzione per la gestione delle interruzioni ad alto livello. Nel caso si stia utilizzando anche le interruzioni a bassa priorità, la direttiva `#pragma interrupt` deve essere sostituita con `#pragma interruptlow` ovvero con il `low` finale. In ultimo viene dichiarata la funzione per la gestione delle interruzioni:

```
void Low_Int_Event (void) {  
    // Programma per la gestione dell'interruzione  
}
```

Come detto, al suo interno, dal momento che tale funzione viene richiamata per la gestione

---

<sup>103</sup> Il Compilatore C18 non cercherà di ottimizzare mai il codice che viene scritto in assembler. Presuppone infatti che questo sia già ottimizzato.

---

delle interruzioni da parte di periferiche differenti è necessario controllare i flag degli interrupt associati alle periferiche, per capire quale periferica ha generato l'interruzione. In particolare alla fine della gestione dell'interruzione bisogna riporre a 0 il flag relativo alla periferica che ha generato l'interruzione.

Per una completa trattazione dei vari flag associati alle periferiche disponibili nel PIC che si sta utilizzando si rimanda al relativo datasheet. Si ricorda che in C18 il nome di tali bit è lo stesso utilizzato nel datasheet. In particolare per cambiare il valore del bit bisogna scrivere:

```
NOME_REGISTRObits.nomeflag
```

Per esempio per modificare il bit GIE presente nel registro INTCON si scriverà:

```
INTCONbits.GIE = 1;
```

Vediamo un esempio in cui si voglia gestire la pressione del tasto BT1 della scheda Freedom II collegato alla linea RB4 del PIC. In particolare si gestirà tale interruzione in modalità compatibile con i PIC16, ovvero facendo uso dell'interrupt vector posto all'indirizzo 0x08 (alta priorità).

```
#include <p18f4550.h>
#include <portb.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilitato il watchdog timer
//LVP = OFF          Disabilitato programmazione LVP
//PBADEN = OFF       Disabilitato gli ingressi analogici

// Prototipo di funzione
void High_Int_Event (void);

//Interrupt vector per modalità compatibile
#pragma code high_interrupt_vector = 0x08

void high_interrupt (void) {

    // Salto per la gestione dell'interrupt
    __asm GOTO High_Int_Event __endasm
}

#pragma code

#pragma interrupt High_Int_Event

// Funzione per la gestione dell'interruzione
void High_Int_Event (void) {

    // Indice per il ciclo di pausa
    int i;

    // Controllo che l'interrupt sia stato generato da PORTB
    if (INTCONbits.RBIF == 1 ) {
```

```

        //pausa filtraggio spike
        for (i=0; i<10000; i++){

        }

        // Controllo la pressione di RB4
        if (PORTBbits.RB4 == 0) {

                // Inverto lo stato del LED 0
                LATDbits.LATD0 = ~LATDbits.LATD0;

        }

        // Resetto il flag d'interrupt per permettere nuove interruzioni
        INTCONbits.RBIF = 0;

    }
}

void main (void){

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Abilita i resistori di pull-up sulla PORTB
    EnablePullups();

    // Abilito le interruzioni su PORTB
    INTCONbits.RBIE = 1;

    // Abilito modalità compatibile (di default vale già 0)
    RCONbits.IPEN = 0;

    // Abilito l'interrupt globale
    INTCONbits.GIE = 1;

    // Abilito l'interrupt periferiche
    INTCONbits.PEIE = 1 ;

    // Ciclo infinito
    while(1){

    }

}

```

---

...i programmi cominciano ad essere più complessi...ma questo significa che ci si comincia a divertire di più! ...dunque abbiamo un'ottima ragione per andare avanti.

In questo esempio si vede che è possibile gestire la pressione di un pulsante senza che il PIC stia di continuo a leggere il valore del pin. In particolare la funzione `main` è praticamente uguale alla funzione `main` degli altri progetti. Unica differenza è che prima di entrare nel ciclo `while` a non far nulla, si impostano ed abilitano le interruzioni. Come al solito per poter utilizzare i pulsanti sulla scheda Freedom II è necessario abilitare i resistori di pull up interni al PIC:

```
EnablePullups();
```

Dopo aver impostato i resistori di pull-up si abilitano le interruzioni associate ai pin della PORTB, ovvero settando il bit RBIE del registro INTCON:

```
INTCONbits.RBIE = 1;
```

Una volta impostato l'hardware per la gestione delle periferiche, vengono abilitate le interruzioni:

```
// Abilito modalità compatibile (di default vale già 0)
RCONbits.IPEN = 0;

// Abilito l'interrupt globale
INTCONbits.GIE = 1;

// Abilito l'interrupt periferiche
INTCONbits.PEIE = 1 ;
```

Fatto questo il programma entra in un ciclo infinito `while (1)` e non fa nulla. In applicazioni reali si potrebbe mandare in stato di Sleep il microcontrollore in modo da risparmiare potenza. Si ricorda infatti che le interruzioni sono tra gli eventi che permettono di risvegliare il PIC dallo stato di Sleep. Oltre a dormire, il PIC potrebbe anche svolgere altre operazioni, quali per esempio fare il polling su altre periferiche...o fischiare!

Per quanto riguarda la parte iniziale del codice si noti che l'interrupt vector è stato posto a 0x08, ovvero in modalità compatibile. Inoltre si è fatto uso della direttiva:

```
#pragma interrupt
```

piuttosto che della direttiva:

```
#pragma interruptlow
```

Questo discende dal fatto che stiamo gestendo le interruzioni in modalità compatibile, ovvero l'equivalente della priorità alta. La funzione per la gestione dell'interrupt è la seguente:

```
// Indice per il ciclo di pausa
int i;

// Controllo che l'interrupt sia stato generato da PORTB
if (INTCONbits.RBIF == 1 ) {

    //pausa filtraggio spike
    for (i=0; i<10000; i++){
```

```

    }

    // Controllo la pressione di RB4
    if (PORTBbits.RB4 == 0) {

        // Accendo il LED 0
        LATDbits.LATD0 = ~LATDbits.LATD0;

    }

    // Resetto il flag d'interrupt per permettere nuove interruzioni
    INTCONbits.RBIF = 0;

}

```

Inizialmente si è dichiarata la variabile `i` utilizzata come nostro solito per un filtraggio degli spike. Dunque non cambia nulla da una normale funzione. In una funzione per la gestione delle interruzioni si possono anche utilizzare variabili globali, ma in questo caso si deve avere l'accortezza di dichiarare la variabile globale di tipo `volatile`. Se per esempio si dichiarasse la variabile globale `temperatura`, normalmente si scriverebbe fuori dalla funzione `main`:

```
int temperatura = 0;
```

Questa variabile così definita potrebbe essere utilizzata dalla funzione `main` come anche da qualunque altra funzione. Qualora tale variabile fosse utilizzata dalla funzione di gestione delle interruzione (in qualunque priorità o modalità) eventuali assunzioni che il compilatore potrebbe fare per ottimizzare il codice non varrebbero più, poiché la variabile `temperatura` potrebbe essere variata in qualunque momento dalla funzione d'interrupt. Per tale ragione, al fine di mettere in guardia il compilatore e obbligarlo a non fare assunzioni di alcun tipo nei confronti della variabile d'interesse, bisogna dichiarare la variabile in questo modo.

```
volatile int temperatura = 0;
```

Dopo questo appunto, vediamo cosa viene fatto all'interno della funzione di gestione dell'interrupt. Come prima cosa si controlla se l'interrupt è stato generato dalla pressione di un pulsante ovvero dalla variazione di stato dei bit della `PORTB`. Per fare questo si controlla il bit `RBIF` del registro `INTCON`. In questo caso, essendo `PORTB` l'unico hardware abilitato ad interrompere non sarebbe in realtà necessario fare questo controllo, ma per chiarezza e generalizzazione abbiamo fatto tale test.

```
if (INTCONbits.RBIF == 1 )
```

Una volta assodato che l'interrupt è stato generato dalla pressione di un pulsante, viene posto un filtro per eliminare gli spike, semplicemente facendo un conteggio a vuoto. Un conteggio di questo tipo viene detto bloccante, poiché durante il conteggio non è possibile fare altre cose. Nei casi in cui si voglia fare una pausa non bloccante si potrebbe far uso dei Timer interni e di altri artifici.

Dopo la pausa viene controllato se il pulsante è ancora premuto, qualora dovesse essere ancora premuto il pin `RD0`, collegato con il LED 0, viene invertito di stato per mezzo dell'operatore bitwise `~`.

```
LATDbits.LATD0 = ~LATDbits.LATD0;
```

---

Dunque se il LED è spento viene acceso, mentre se è acceso viene spento. Una volta svolta l'operazione di gestione dell'interrupt viene resettato il bit RBIF.

```
INTCONbits.RBIF = 0;
```

Si noti che tale istruzione appartiene al blocco `if` e non alla funzione principale. Questo è fondamentale qualora si gestiscano anche altre interruzioni. Infatti si pulissero tutti i flag d'interruzione alla fine della funzione di gestione dell'interruzione, qualora una periferica avesse generato un interrupt durante l'esecuzione di istruzioni associate ad un'altra interruzione, tale informazione verrebbe persa; dunque ogni flag deve essere riposto a 0 all'interno del blocco `if` che ha controllato il suo stato. A fine gestione di un interrupt, qualora qualche altra periferica avesse richiesto un interrupt durante la gestione di un altro interrupt, potrà essere gestito.

Dal momento che non sono ancora note molte periferiche interne al PIC non si farà alcun esempio sulla gestione delle interruzioni abilitando sia l'alta che la bassa priorità. Esempi con ambedue le modalità attive verranno comunque mostrati nei prossimi Capitoli.

# Capitolo VII

## Scriviamo una libreria personale

Dopo aver appreso le basi della programmazione in C ed in particolare i dettagli per applicare tale linguaggio di programmazione nell'ambiente di sviluppo MPLAB, vediamo come organizzare i nostri programmi per mezzo delle librerie. Il Capitolo inizierà con una breve panoramica per spiegare l'esigenza di una libreria per poi entrare in qualche dettaglio nella spiegazione delle direttive, frequentemente utilizzate all'interno delle librerie. In ultimo verrà mostrato un esempio pratico al fine di capire meglio come effettuare la compilazione di una libreria.

### Perché scrivere una libreria

Abbiamo già visto che ogni programma C può svilupparsi su di un unico file creando un mostro o in maniera più pratica si era già suggerito di dividere ed organizzare programmi complessi facendo uso di più file. Il file esterno poteva essere poi incluso all'interno del progetto per mezzo della direttiva `#include` o per mezzo della finestra dei file di progetto. Questa tecnica può essere utilizzata senza problemi qualora si abbiano poche funzioni specifiche per un'applicazione, ma quando le funzioni sono specifiche per un determinato dispositivo o famiglia di applicazioni, ovvero possono tornare utili in molteplici situazioni, è bene creare una libreria, ovvero una raccolta di funzioni scritte solo ed esclusivamente per un determinato dispositivo. Un esempio potrebbe essere una libreria per gestire display LCD o una comunicazione RS232 e via dicendo.

Si capisce subito che l'utilità di una libreria sta nel fatto che fornisce tutte le funzioni (o almeno questa è sempre la speranza) di cui avremo bisogno per gestire la nostra periferica. Se la libreria è ben fatta permetterà anche di raggiungere un livello di astrazione per l'utilizzo della periferica tale per cui non bisogna sapere molto della periferica stessa.

Oltre a raccogliere le funzioni una libreria ha l'utilità di rappresentare un file comune a diverse applicazioni, permettendo di aggiornare un unico file qualora dovesse essere trovato un errore o si volessero aggiungere nuove funzioni. Se infatti si dovesse creare per ogni programma un file c con le funzioni per la gestione di un display LCD, semplicemente copiando ed incollando le funzioni da un altro progetto precedentemente scritto, si avrebbero due copie delle funzioni. Facendo ulteriori copia ed incolla, il numero di copie delle nostre funzioni diventerebbe tale che se si volesse modificare una sola funzione dovremmo andare a modificare uno per uno i sorgenti dei vari progetti. Creando una libreria si ha un'unica raccolta che tutti i progetti possono includere, dunque ogni modifica si ripercuote automaticamente su ogni progetto. Nonostante si possa modificare la libreria in un solo punto, si avrà comunque la necessità di ricompilare ogni progetto che fa uso della libreria.

Un ultimo aspetto associato all'utilizzo delle librerie, e questo vale in generale, come anche gli altri punti, è quello di permettere di mantenere il codice sorgente nascosto. Come si vedrà per utilizzare una libreria si avrà infatti bisogno solo del file header .h con i prototipi delle funzioni e il file .lib ovvero il file oggetto con il codice compilato. Il file sorgente .c non è



---

necessario poiché compilato nel file `.lib`. In questo modo molte società nascondono il file sorgente permettendo di implementare librerie ottimizzate che i competitori avranno difficoltà a copiare. Nel caso di semplici applicazioni osservando il codice assembler è possibile risalire più o meno al codice originale.

## Le direttive

Come detto una direttiva non rappresenta un'istruzione che il compilatore tradurrà in linguaggio macchina, bensì una guida per il compilatore o meglio pre-compilatore (ovvero colui che viene prima del compilatore) che serve per capire come organizzare la compilazione. Abbiamo già incontrato alcune direttive che per completezza verranno ripetute in maniera da avere una lista più completa. Le direttive che verranno descritte sono solo alcune di quelle disponibili, in particolare sono le più usate. Ogni direttiva come già visto è introdotta dal carattere `#` e non termina con il solito punto e virgola necessario per le istruzioni C. Vediamo le direttive:

- **#include**

La direttiva `#include` permette di includere un file esterno all'interno del file in cui viene dichiarata. La sua sintassi è differente a seconda del percorso a cui si vuol far riferimento.

```
#include <nome_file>
```

Questo formato viene utilizzato per includere file, siano essi `.c` `.h` o altri formati in cui siano scritti contenuti riconoscibili dal compilatore. Il file viene ricercato tra i percorsi di ricerca impostati nelle *Build Options*, alla voce *Include Search Path*. Tra questi percorsi vi è quello standard `../MCC18/h` relativo al percorso d'installazione delle librerie del C18. Qualora si voglia fare riferimento ad un percorso assoluto o alla directory dove si trova anche il nostro progetto, si può far uso del formato:

```
#include "percorso_file/nome_file"
```

La parte del percorso file può anche rimossa, qualora non sia abbia interesse ad un percorso assoluto. In questo caso i percorsi che verranno utilizzati per la ricerca del file sono prima la directory corrente e successivamente gli *Include Search Path*. In generale si sconsiglia di utilizzare i percorsi assoluti in modo da rimanere più flessibili in termini di salvataggio ed esecuzione del progetto.

- **#define**

La direttiva `#define`, può essere utilizzata per vari scopi. Normalmente viene utilizzata per definire delle costanti o assegnare un nome particolare ad un pin del PIC.

```
#define MAX_VALUE 12  
  
#define LED LATDbits.LATD4
```

Nel primo esempio si definisce la costante `MAX_VALUE` pari al valore 12.

---

L'utilizzo della costante `MAX_VALUE` permetterà di utilizzare tale nome in tutti i punti che si vuole, ma nel caso in cui si dovesse cambiare il suo valore basterà cambiare la riga di definizione.

Nel secondo esempio si assegna il nome `LED` al pin `LATDbits.LATD4`. In questo modo qualora si voglia accendere il LED si potrà scrivere

```
LED = 0x01;
```

piuttosto che:

```
LATDbits.LATD4 = 0x01;
```

Questo risulta molto utile nei casi in cui la posizione del LED potrebbe cambiare. Infatti cambiano solo la riga in cui viene definita il nome `LED`, ogni parte del programma saprà dove è posizionato il LED.

La direttiva può essere anche utilizzata per definire semplicemente un nome senza dare un valore:

```
#define LED
```

Questo risulta utile nel controllo che verrà spiegato nello spiegare la direttiva `#ifndef`.

- **`#ifndef`**

La direttiva `#ifndef` permette di far compilare il codice che segue, qualora una determinata costante non sia stata definita. Il codice che deve essere compilato deve essere terminato dalla direttive `#endif`. Questa funzione risulta molto utile per evitare di includere due volte uno stesso file. Qualora un file venisse incluso due volte si ha infatti il problema di codice duplicato ed in particolare di nomi di funzioni e variabili duplicate. Questo creerà molti errori di compilazione. La direttiva `#ifndef` permette invece di controllare ed evitare inclusioni multiple.

```
#ifndef LIBRERIA_LCD
#define LIBRERIA_LCD

// Codice della mia libreria o funzioni

#endif
```

Dall'esempio si può vedere che la direttiva effettua un test sull'esistenza o meno del nome `LIBRERIA_LCD`. Se questa non risulta essere stata dichiarata da nessun'altra parte vuol dire che il file non è stato mai incluso prima e dovrà dunque essere compilato. Alla fine del controllo viene definito il nome `LIBRERIA_LCD` in modo tale che includendo una seconda volta il file da qualche altra parte, questo non verrà più compilato.

Dopo aver definito il nome d'interesse per il controllo del nostro codice, è possibile inserire il codice che verrà compilato. Il termine del codice da compilare deve essere segnalato dalla direttiva `#endif`.

Il blocco `if` appena introdotto può essere ripetuto numerose volte all'interno del

---

nostro programma, creando in questo modo una compilazione ottimizzata a seconda delle impostazioni esterne o definite dall'utente.

- **#endif**

Si veda la direttiva `#ifndef`.

- **#warning**

La direttiva `#warning` permette di aggiungere dei messaggi di warning al nostro codice.

```
#warning Questo è il messaggio che visualizzo
```

Il messaggio di warning può essere utilizzato qualora si voglia avvisare l'utilizzatore di una libreria di un potenziale problema. Come gli altri messaggi di warning è bene fare in modo che un eventuale messaggio visualizzato da una libreria venga propriamente compreso ed eliminato in modo opportuno.

- **#error**

La direttiva `#error` può essere utilizzata per generare un errore ed evitare che la compilazione venga portata a termine. Un esempio è il seguente.

```
#error Questo è il messaggio che visualizzo per l'errore
```

può ritornare particolarmente utile qualora un parametro fondamentale non sia del valore giusto o non sia stato definito. In questo caso è bene che la compilazione venga interrotta visto che non è possibile garantire la corretta esecuzione del programma stesso.

## Esempio di creazione di una Libreria

La realizzazione di una libreria non è molto complicata, e alla luce delle nuove conoscenze si hanno tutti gli strumenti per realizzarne di professionali. In questo breve paragrafo non verrà spiegata la fase di ingegnerizzazione che precede la realizzazione della libreria. In ogni modo si presume che si abbia buona conoscenza dell'integrato o sistema per il quale si voglia realizzare la libreria. Dopo la conoscenza tecnica è bene organizzare il lavoro per quanto riguarda le funzioni da implementare e il modo con cui implementarle. Quando tutto questo è chiaro è possibile iniziare la realizzazione di una libreria. Nel seguente esempio si scriverà una semplice libreria che permette di scrivere dati all'interno della EEPROM contenuta all'interno di ogni PIC18. Per una conoscenza tecnica sulla memoria EEPROM si rimanda al paragrafo in cui si è introdotta l'architettura dei PIC18. Per tale libreria si scriveranno due semplici funzioni una per scrivere un dato nella EEPROM e una per leggere un dato nella EEPROM.

Per realizzare una libreria si procede come per un progetto normale anche se poi le

informazioni sul PIC del progetto non verranno effettivamente utilizzate. Il nome del progetto deve essere quello che verrà dato alla libreria, nel nostro caso intEEPROM. Infatti il nome della libreria .lib che verrà generata dopo la compilazione avrà il nome del progetto stesso. Si fa presente che è buona pratica avere un'unica cartella dove verranno salvati tutti i progetti di libreria. Successivamente bisogna creare un file.c con il nome della nostra libreria, nel nostro caso intEEPROM.c e un file .h (noto come header file) intEEPROM.h.

Nel file .c viene scritto il codice vero e proprio con cui implementiamo le nostre funzioni, mentre nel file .h vengono scritti solo i prototipi delle funzioni con loro descrizione. Normalmente per sapere come utilizzare una libreria si fa infatti riferimento al suo header file. Nel nostro caso si ha:

```
#include <p18cxxx.h>

#ifndef FLAG_intEEPROM
#define FLAG_intEEPROM

/*****
 *
 * Description: This function writes a byte inside
 *              the EEPROM.
 *
 * Parameters:
 *
 * data: byte to write
 * address : Address where the byte must be written
 *
 * Return:
 *
 * char: 1: The byte has been properly written
 *       0: The byte has not been properly written
 *
 * Note:
 *
 * During the writing process the Interrupts are
 * disabled. But the old status is restored.
 *
 *****/
char writeIntEEPROM (unsigned char data, unsigned char address);

/*****
 *
 * Description: This function read a byte inside
 *              the EEPROM.
 *
 * Parameters:
 *
 * address : Address where the byte must be read
 *
 * Return:
 *
 * unsigned char: It returns the data byte
 *
 *****/
unsigned char readIntEEPROM (unsigned char address);

#endif
```

---

Dal momento che nella libreria si fa uso di nomi di registri interni al PIC è necessario includere il file:

```
#include <p18cxxx.h>
```

Questo file rappresenta un file generico per mezzo del quale verrà incluso il file del PIC relativo al progetto. Se non si includesse tale file i registri per la gestione della memoria EEPROM non verrebbero riconosciuti. Successivamente si è fatto uso delle direttive:

```
#ifndef FLAG_intEEPROM  
#define FLAG_intEEPROM
```

Questo permette di avere multi inclusioni dello stesso file, senza avere conflitti ed errori di compilazione. Si noti che ogni funzione è preceduta da un commento in cui si riporta una breve descrizione della mansione svolta dalla funzione e la descrizione dei parametri che bisogna passare e che vengono ritornati. In questo modo l'utilizzatore ha tutte le informazioni necessarie per un suo corretto utilizzo. Qualora la funzione abbia degli effetti collaterali questo è il posto giusto per scriverli. La dichiarazione delle due funzioni è seguita dalla direttiva `#endif`.

Nel file `.c` viene scritto il sorgente vero e proprio. In questo file diversamente dal file `.h` non vengono normalmente scritte le informazioni associate alla funzione. Infatti se si fa uso del file di libreria che andremo a generare, l'utilizzatore potrebbe non avere il file sorgente. Il file `.c` rappresenta un file sorgente tradizionale, dove si dichiarano le relative funzioni scritte nell'header file. Nel nostro caso si ha:

```
#include "intEEPROM.h"  
  
/*****  
// write function implementation  
*****/  
  
char writeIntEEPROM (unsigned char data, unsigned char address) {  
  
    // Flag used to store the GIE value  
    unsigned char flagGIE = 0;  
  
    // Flag used to store the GIEH value  
    unsigned char flagGIEH = 0;  
  
    // Flag used to store the GIEL value  
    unsigned char flagGIEL = 0;  
  
    // Set the address that will be written  
    EEADR = address;  
  
    // Set the data that will be written  
    EECON1bits.EEPGD = 0;  
  
    // EEPROM memory is pointed  
    EECON1bits.CFGS = 0;
```

```

// Enable write
EECON1bits.WREN = 0x01;

// Check and store the Interrupt Status
if (INTCONbits.GIE == 1) {
    INTCONbits.GIE = 0;
    flagGIE = 1;
}

if (INTCONbits.GIEH == 1) {
    INTCONbits.GIEH = 0;
    flagGIEH = 1;
}

if (INTCONbits.GIEL == 1) {
    INTCONbits.GIEL = 0;
    flagGIEL = 1;
}

// Start the writing enabling sequence
EECON2 = 0x55;
EECON2 = 0xAA;

// Initiate writing process
EECON1bits.WR = 0x01;

// Wait the end of the writing process
while (EECON1bits.WR);

// Restore the previous interrupt status
if (flagGIE == 1) {
    INTCONbits.GIE = 1;
}

if (flagGIEH == 1) {
    INTCONbits.GIEH = 1;
}

if (flagGIEL == 1) {
    INTCONbits.GIEL = 1;
}

// Disable the writing process
EECON1bits.WREN = 0x00;

// Check if the data has been properly written,
// a simple read back is done
if (readIntEEPROM (address) == data) {

    return (1);

} else {

    return (0);
}
}

```

```

//*****
//                               Read Function Implementation
//*****

unsigned char readIntEEPROM (unsigned char address) {

    unsigned char data = 0;

    // Set the memory address that will be read
    EEADR = address;

    // EEPROM memory is pointed
    EECON1bits.EEPGD = 0;

    // EEPROM access enable
    EECON1bits.CFGS = 0;

    // Initiate reading
    EECON1bits.RD = 0x01;

    // Data is read from the register
    data = EEDATA;

    return (data);
}

```

Si noti che il file C non necessita di nessuna direttiva per evitare la multi inclusione, visto che il file .c non verrà affatto incluso. L'unica direttiva di cui si fa normalmente uso è:

```
#include "intEEPROM.h"
```

Questo permette di includere il file header nel quale sono stati dichiarati i prototipi di funzione e si sono inclusi gli altri file necessari alla compilazione. Ogni altro file necessario deve essere incluso nel file di header, ed indirettamente verrà incluso nel file sorgente. Il non seguire questa pratica non crea nessun problema, ma è un modo ordinato per mostrare i file inclusi in un solo punto.

Il programma in C ripete i passi già spiegati quando è stata introdotta la memoria EEPROM. Unica differenza sta nella funzione di scrittura in cui si è gestito il caso di disabilitazione delle interruzioni. Per chiarezza si è gestito il caso alta, bassa e compatibile PIC16, anche se i bit del registro INTCON sono in realtà gli stessi. La funzione di write, effettua anche un controllo sul byte scritto, in maniera da permettere all'utilizzatore di effettuare un controllo veloce sulla corretta scrittura della EEPROM.

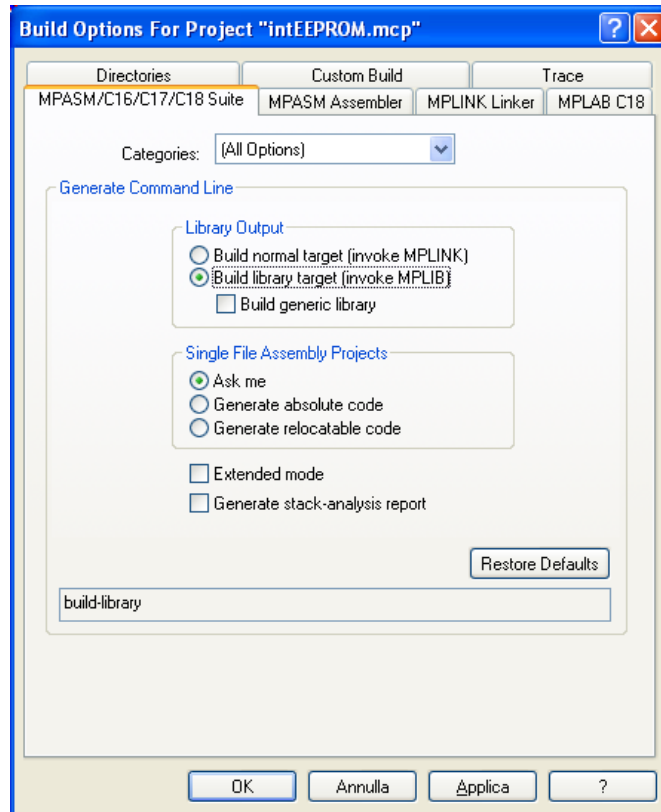
Arrivato a questo punto compiliamo il nostro programma come abbiamo sempre fatto...  
...ed otterremo un errore!

*Error - could not find definition of symbol 'main' in file 'C:\MCC18\lib\c018i.o'.*

Questo errore è legato al fatto che non abbiamo dichiarato la funzione `main`, che come detto deve essere sempre dichiarata. In realtà in una libreria non serve la funzione `main`, la quale deve dichiarata nel programma principale che farà uso della libreria .

La ragione per cui viene generato l'errore è dovuta al fatto che per ottenere una libreria è necessario impostare il compilatore in maniera differente. In particolare deve essere abilitato il programma *mplib*. Per fare questo bisogna andare tra le *Build Options*, al Tab *MPASM* e selezionare l'opzione *Build Library target* invocando *mplib*, come riportato in Figura 51.

Normalmente l'opzione abilitata è *Build Normal target* per mezzo di *mplink*. La libreria viene invece creata da *mplib* partendo dal file oggetto creato dal compilatore. *mplab* permette anche di includere in una sola libreria più file oggetto, gestendo il tutto da riga di comando (shell DOS). Una volta abilitato *mplib*, riprovando a compilare il sorgente si ha che il tutto va a buon fine. Andando nella directory del nostro progetto, potremo vedere che è stato creato il file *intEEPROM.lib* ovvero la nostra libreria. Diversamente dai casi precedenti non è presente alcun file *.hex*.



**Figura 51:** Abilitazione di *MPLIB* per creare una libreria

## Utilizziamo la nostra Libreria

Vediamo ora un semplice esempio in cui viene utilizzata la libreria appena creata:

```
#include <p18f4550.h>
#include <intEEPROM.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS           Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici
```



```

void main (void) {

    unsigned char data = 0xAA;
    unsigned char address = 0;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Scrivo e visualizzo il dato dalla EEPROM interna

    if (writeIntEEPROM (data, address) == 1) {

        LATD = readIntEEPROM (address);

    } else {

        LATD = 0xFF;
    }

    while(1) {

    }

}

```

E' possibile notare che il file di libreria è stato incluso per mezzo della direttiva `#include`. Per poter però compilare propriamente il programma è necessario impostare i percorsi dove andare a cercare l'header file della nostra libreria. In particolare è necessario impostare i percorsi delle *Build Options*, includendo la cartella in cui è presente l'header file.

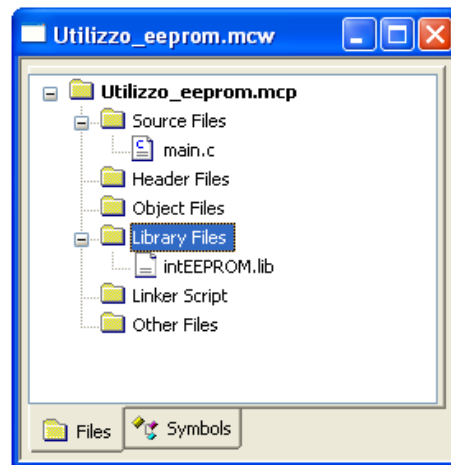
#### **Nota:**

*Normalmente negli ambienti di sviluppo più noti bisogna impostare anche il percorso di libreria per completare il tutto. Ho notato che impostando il percorso dove trovare il file .lib la compilazione non va comunque a buon fine. Per permettere al compilatore di trovare il file di libreria bisogna includere la nostra libreria all'interno della cartella Library Files del nostro progetto. Come riportato in Figura 52. Personalmente, dal momento che i file di libreria standard della Microchip richiedono solo d'impostare il percorso non ho ben capito se è un problema o ho fatto qualche passo sbagliato. Qualora avessi sbagliato qualche passo devo dire che l'impostazione non è conforme agli standard sia delle librerie Microchip sia di altri ambienti di sviluppo.*

Una volta inclusa la libreria è possibile compilare con successo il nostro programma di

---

esempio. Si noti che nonostante il problema d'inclusione della libreria, il file c non è stato esplicitamente incluso in nessun punto.



**Figura 52:** *Inclusione del file di libreria .lib*

Si osservi che nel programma di esempio si è sfruttato il valore di ritorno della funzione di scrittura, per sapere se il byte è stato scritto con successo. Tale controllo è particolarmente utile nonché fondamentale in ogni applicazione professionale.

# Capitolo VIII

## Utilizziamo i Timer interni al PIC

Dopo tanto peregrinare tra un'istruzione C e l'altra siamo finalmente giunti alla descrizione dell'Hardware interno del PIC. Quanto verrà descritto nei prossimi Capitoli vi permetterà di utilizzare il PIC in applicazioni sempre più professionali e divertenti. In questo Capitolo vengono introdotti i Timer interni al PIC, focalizzando l'attenzione sul Timer0. Gli altri Timer verranno solo accennati visto che sono spesso associati a funzioni speciali. In particolare il Timer2 verrà descritto in maggior dettaglio quando si parlerà del PWM. Il Capitolo ha inoltre la funzione di completare l'argomento sugli Interrupt, visto che negli esempi verranno finalmente utilizzati i due livelli d'interrupt.

### Descrizione dell'hardware e sue applicazioni

Giunti a questo punto siamo sempre più consapevoli che per mezzo di un microcontrollore è possibile fare molte cose in contemporanea o meglio in successione ma con l'apparenza che siano svolte contemporaneamente. Per raggiungere questo è necessario organizzare il programma e sfruttare le interruzioni, dove necessario, per poter creare quella fluidità di esecuzione che crea quell'apparente esecuzione Real Time, ovvero di esecuzione in tempo reale. In tale contesto la dicitura Real Time potrebbe essere mal intesa; in ambito professionale una semplice fluidità del programma che non crea disagio all'utilizzatore, viene in generale definita soft Real Time. Si pensi per esempio al mouse del PC, questo normalmente si muove in maniera fluida e non crea disagio all'utilizzatore; ciononostante non è inusuale che in alcuni casi si muova a scatti. Per tale ragione il mouse non è realmente una periferica Real Time, ed in particolare il software che la gestisce risponde alla definizione di soft Real Time. Periferiche Real Time o meglio hard Real Time sono quelle periferiche il cui software permette di eseguire determinati compiti in un tempo stabilito (definito in gergo time budget). Qualunque cosa accada questo tempo verrà rispettato garantendo che le azioni o risultati derivanti dal servizio prestato dal software siano sempre forniti in tempo utile e in forma corretta. Tra le tecniche per raggiungere tali livelli di performance vi è l'utilizzo degli interrupt<sup>104</sup> che permette ad una periferica importante di interrompere la normale esecuzione del programma. Una seconda tecnica utilizzata è per mezzo di Timer, che permettono di assegnare determinati tempi di esecuzione a dei processi o funzioni<sup>105</sup>.

I Timer interni al PIC sono effettivamente quello che ci si aspetta che siano ovvero dei sistemi per contare. All'interno dei PIC18 sono presenti fino a 6 Timer a seconda del modello del PIC utilizzato. Nel PIC18F4550 sono presenti 4 Timer nominati Timer0, Timer1, Timer2, Timer3. Il Timer0 rappresenta quello per uso generico e sarà quello descritto in maggior dettaglio nei paragrafi ed esempi che seguiranno. Gli altri Timer anche se è possibile utilizzarli per applicazioni generiche, sono spesso utilizzati per applicazioni specifiche.

<sup>104</sup> I tempo di latenza è uno dei parametri fondamentali nell'analisi del time budget.

<sup>105</sup> I Timer vengono spesso utilizzati nei sistemi operativi per la realizzazione dello scheduling dei processi. Nel seguente Capitolo non si affronteranno le problematiche associate a questo utilizzo poiché la trattazione richiede conoscenze che vanno oltre la semplice lettura di un datasheet o un libro.

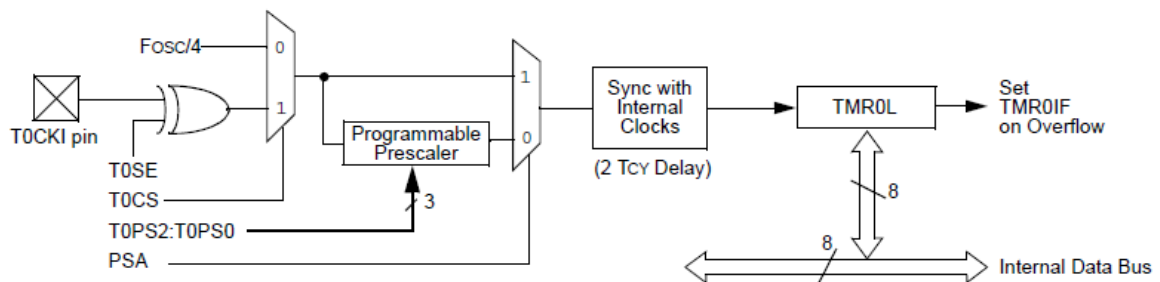
Per esempio il Timer1 può essere utilizzato come oscillatore secondario ed in particolare possedendo la circuiteria per far oscillare un quarzo a bassa frequenza, può essere utilizzato per far oscillare un quarzo da 32768KHz, utilizzato per la realizzazione di un Real Time Clock Calendar.

Il Timer2 e Timer3 sono spesso associati con i moduli CCP utilizzati per la generazione del segnale PWM. Quando un Timer viene associato ad una periferica non può essere utilizzato in applicazioni generiche. Vediamo in maggior dettaglio come funziona il Timer0, la cui comprensione porterà facilmente alla comprensione degli altri Timer, ma per i quali si rimanda al datasheet del PIC utilizzato.

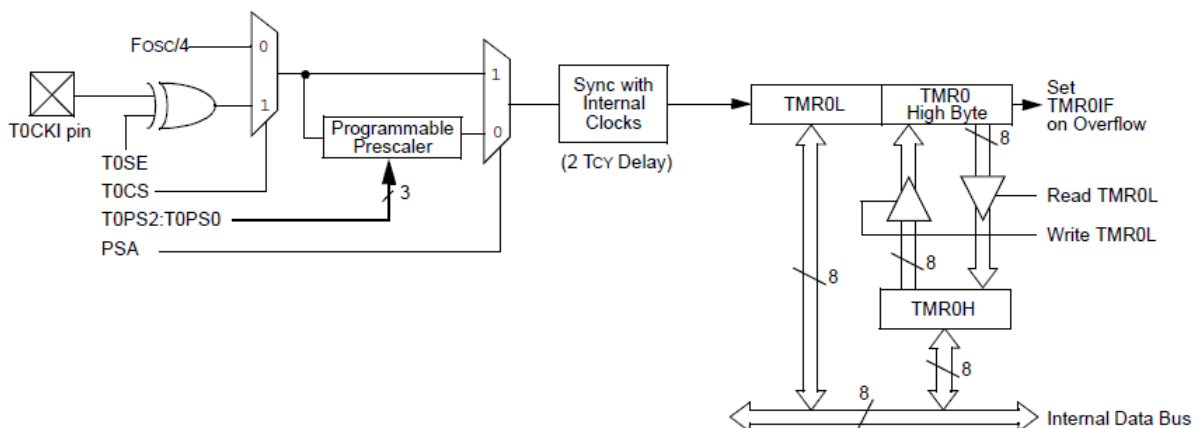
### Descrizione Timer0

- Il Timer può lavorare in modalità ad 8 o 16 bit
- I registri del Timer possono essere sia letti che scritti
- Possibilità di utilizzare il Prescaler
- Sorgente di Clock selezionabile (interna od esterna)
- Selezione del fronte dell'incremento
- Possibilità di generare un interrupt

In Figura 53 è riportato lo schema a blocchi del modulo Timer0 in modalità ad 8 bit mentre in Figura 54 è riportato lo schema a blocchi in modalità a 16 bit.



**Figura 53:** Schema a blocchi del Timer0 in modalità 8 bit



**Figura 54:** Schema a blocchi del Timer0 in modalità 16 bit

Dallo schema a blocchi è subito possibile vedere che sulla sua sinistra è possibile selezionare la sorgente del clock che permetterà l'incremento del contatore. In particolare è

possibile selezionare una sorgente esterna ovvero entrante dal pin TOKI o la sorgente interna Fosc/4 ovvero la frequenza di esecuzione delle istruzioni. Successivamente è presente un percorso diretto verso un secondo multiplexer di selezione, ed un percorso attraverso un prescaler ovvero un divisore di frequenza. Questo permette rallentare la frequenza principale, sia essa quella interna od esterna. La divisione può essere selezionata da 2 a 256 con passi di potenze di 2, ovvero, 2, 4, 8, 16, 32, 64, 128, 256. Successivamente è presente una linea di ritardo di due cicli di clock; questo permette di sincronizzare il segnale con il clock interno.

Il clock che si viene a generare dopo questo percorso andrà a far incrementare il contatore Timer0, sia che sia impostato a 8 bit ovvero con conteggio massimo fino a 256, sia che sia impostato in modalità a 16 bit, ovvero con conteggio massimo fino a 65535. Lo schema a blocchi a 16 bit risulta leggermente più complesso poiché il secondo registro associato al Timer0, ovvero TMR0H, deve essere scritto e letto in maniera opportuna in maniera da garantire il corretto funzionamento del Timer stesso. Si fa notare che al fine di garantire la corretta esecuzione temporale dei tempi impostati, il prescaler viene azzerato ogni volta che si scrive nei registri del Timer0. Il suo Reset non cambia però il rapporto di divisione che è stato impostato.

## I registri interni per il controllo del Timer0

Vediamo ora come impostare il Timer0 al fine di farlo funzionare. Il registro principale associato alla sua impostazione è il T0CON, il significato di ogni bit è il seguente:

### Registro T0CON: Timer0 Control Register

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Leggenda							
R = Readable bit		W = Writable bit		U = Unimplemented bit read as 0		S : Settable bit	
-n = Value at POR		1 = Bit is set		0 = Bit is cleared		x = Bit is unknown	

<b>Bit 7</b>	<b>TMR0ON</b> : Timer0 ON/OFF bit 1 : Abilita il Timer0 0 : Disabilita il Timer0
<b>Bit 6</b>	<b>T08BIT</b> : Timer0 8-Bits/16Bits Control bit 1 : Il Timer0 è configurato ad 8 bit 0 : Il Timer0 è configurato a 16 bit
<b>Bit 5</b>	<b>T0CS</b> : Bit di selezione del Clock 1 : Transizione sul pin T0CKI 0 : Clock d'istruzione interna
<b>Bit 4</b>	<b>T0SE</b> : Selezione del fronte del Timer0 1 : Incrementa sul fronte di discesa del pin T0CKI 0 : Incrementa sul fronte di salita del pin T0CKI
<b>Bit 3</b>	<b>PSA</b> : Bit di assegnamento del Prescaler 1 : Il Prescaler non è assegnato al Timer0 0 : Il Prescaler è assegnato al Timer0
<b>Bit 2-0</b>	<b>T0PS2-T0PS0</b> : 111 : 1:256 Valore del Prescaler 110 : 1:128 Valore del Prescaler 101 : 1:64 Valore del Prescaler 100 : 1:32 Valore del Prescaler 011 : 1:16 Valore del Prescaler 010 : 1:8 Valore del Prescaler 001 : 1:4 Valore del Prescaler 000 : 1:2 Valore del Prescaler

Normalmente i Timer vengono fatti lavorare per mezzo di Interrupt, in maniera tale da avere un conteggio senza bloccare il resto del programma. Infatti i Timer sono indipendenti dall'esecuzione del programma, ovvero una volta impostati non richiedono nessun'altra azione da parte del programma. Se sono attivate le interruzioni il Timer interrompe il programma principale e sarà solo a questo punto che il programma deve gestire l'evento associato al Timer.

Per abilitare le interruzioni è necessario abilitare il bit TMR0IE del registro INTCON. All'interno dello stesso registro è anche presente il bit TMR0IF che permette di visualizzare l'evento d'interruzione associato al Timer1. Si ricorda che tale bit deve essere posto a 0 via software al termine della routine di gestione dell'interruzione. In ultimo a seconda della modalità d'interruzione che viene utilizzata bisogna impostare il bit TMR0IP. Bisogna impostare il bit a 0 per abilitarlo ad interruzioni a bassa priorità, mentre deve essere impostato ad 1 per utilizzare le interruzioni ad alta priorità o modalità compatibile. Si ricorda che tale bit di default vale 1, ovvero la periferica è gestita ad alta priorità. Questo permette di ignorare tale bit quando viene utilizzata la modalità compatibile PIC16 che equivale a trattare le interruzioni ad alta priorità, ovvero con l'interrupt Vector impostato ad 0x08.

In Tabella 7 sono riportati i vari registri e bit associati al Timer0.

Registro	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset
<b>TMR0L</b>	Registro Timer0 Low								52
<b>TMR0H</b>	Registro Timer0 High								52
<b>INTCON</b>	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INTIF	RBIF	51
<b>INTCON2</b>	RBPU	INTEDG0	INTEDG1	INTEDG2	-	TMR0IP	-	RBIP	51
<b>T0CON</b>	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	52
<b>TRISA</b>	-	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	54

**Tabella 7:** Registri associati al Timer0

## Esempi di utilizzo del Timer0

Tante parole non valgono nulla se non trattate con un esempio. Iniziamo con un primo esempio in cui si fa semplicemente lampeggiare il LED 0 della scheda Freedom II, facendo uso dei cicli `for` e al tempo stesso si effettui la lettura del pulsante BT1, alla cui pressione si accende il LED 1. Prima di fare brutta figura con il programma che seguirà, anticipo che questo non è il modo esatto di risolvere il problema.

```
#include <p18f4550.h>
#include <portb.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS           Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici
```

```

void main (void){

    // Variabile usata per creare un conteggio fittizio di pausa
    unsigned int i;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Abilita i resistori di pull-up sulla PORTB
    EnablePullups();

    // Ciclo infinito
    while (1) {

        // Pausa
        for (i=0;i<64000; i++) {

        }

        // Controllo del Pulsante BT1
        if (PORTBbits.RB4 == 0) {
            LATDbits.LATD1 = 0x01;
        } else {
            LATDbits.LATD1 = 0x00;
        }

        // Accendo il LED0 per il lampeggio
        LATDbits.LATD0 = 0x01;

        for (i=0;i<64000; i++) {

        }

        // Spengo il LED 0 per il lampeggio
        LATDbits.LATD0 = 0x00;

    }
}

```

Si osservi subito che il pulsante viene letto solo dopo le pause. Caricando il programma sulla scheda Freedom II vi renderete presto conto che la pressione del pulsante non fa accendere e spegnere il LED 1 in maniera molto fluida, visto che sono presenti delle pause più o meno lunghe in cui il programma non effettua il controllo. Un modo per ovviare a tale

limite sarebbe quello di eseguire il controllo all'interno di ogni ciclo di ritardo, in modo da controllare la pressione del pulsante. Alla luce delle nostre esperienze passate e dalla brutta esperienza o brutta figura attuale, sappiamo che un modo professionale per gestire il tutto, è per mezzo delle interruzioni. In questo caso specifico si sarebbe potuto anche evitare, ma se oltre al lampeggio bisogna usare un LCD alfanumerico sul quale scrivere i dati che arrivano dalla porta seriale...leggere la temperatura e controllare che sia suonata la sveglia...in questo caso non c'è altra soluzione snella che l'utilizzo delle interruzioni. Utilizzare le interruzioni permette anche di inserire altre funzionalità senza sconvolgere il programma principale, cosa che nel caso dell'utilizzo dei cicli `for` e letture continue del pulsante, sarebbe stato probabilmente necessario fare.

Vediamo allora un altro esempio, in cui si voglia far lampeggiare il LED 0 e si voglia accendere il LED 1 alla pressione di BT1. Si considera che il lampeggio del LED è a bassa priorità poiché ha solo lo scopo di visualizzare all'utente che il programma è in esecuzione, mentre la pressione del pulsante viene gestita ad alta priorità poiché si considera che alla sua pressione corrisponderà l'accensione del LED 1, che corrisponde all'apertura di una porta d'emergenza.

```
#include <p18f4550.h>
#include <portb.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

//*****
// Dichiarazione Prototipi di funzione
//*****

// Prototipo di funzione per bassa priorità
void Low_Int_Event (void);

// Prototipo di funzione per alta priorità
void High_Int_Event (void);

//*****
// Impostazione e Gestione priorità alta
//*****

#pragma code high_vector = 0x08

void high_interrupt (void) {

    // Salto per la gestione dell'interrupt ad alta priorità
    _asm GOTO High_Int_Event _endasm
}

#pragma code
```



```

#pragma interrupt High_Int_Event

// Funzione per la gestione dell'interruzione ad alta priorità
void High_Int_Event (void) {

    // Indice per il ciclo di pausa
    int i;

    // Controllo che l'interrupt sia stato generato da PORTB
    if (INTCONbits.RBIF == 1 ) {

        //pausa filtraggio spike
        for (i=0; i<10000; i++){

        }

        // Controllo la pressione di RB4
        if (PORTBbits.RB4 == 0) {

            // Accendo il LED 1
            LATDbits.LATD1 = ~LATDbits.LATD1;

        }

        // Resetto il flag d'interrupt per permettere nuove interruzioni
        INTCONbits.RBIF = 0;

    }
}

//*****
// Impostazione e Gestione priorità bassa
//*****

#pragma code low_vector = 0x18

void low_interrupt (void) {

    // Salto per la gestione dell'interrupt a bassa priorità
    __asm GOTO Low_Int_Event __endasm
}

#pragma code

#pragma interruptlow Low_Int_Event

// Funzione per la gestione dell'interruzione a bassa priorità
void Low_Int_Event (void) {

    // Controllo che l'interrupt sia stato generato da Timer0
    if (INTCONbits.TMR0IF == 1 ) {

        // Inverto lo stato del LED 0
        LATDbits.LATD0 = ~LATDbits.LATD0;

        // Resetto il flag d'interrupt per permettere nuove interruzioni
        INTCONbits.TMR0IF = 0;

    }

}

//*****
// Inizio programma principale

```

```

//*****

void main (void){

    // Variabile usata per creare un conteggio fittizio di pausa
    unsigned int i;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0xFF;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    //*****
    // Abilito i pulsanti per le interruzioni ad alta priorità
    //*****

    // Abilita i resistori di pull-up sulla PORTB
    EnablePullups();

    // Abilito le interruzioni su PORTB
    INTCONbits.RBIE = 1;

    // Abilito le interruzioni su PORTB come alta priorità
    INTCON2bits.RBIP = 1;

    //*****
    // Abilito il Timer0 per funzionare a bassa priorità
    //*****

    // Modalità a 16 bit
    T0CONbits.T08BIT = 0;

    // Clock interno
    T0CONbits.T0CS = 0;

    // Abilito Prescaler
    T0CONbits.PSA = 0;

    // Prescaler 32
    T0CONbits.T0PS0 = 0;
    T0CONbits.T0PS1 = 0;
    T0CONbits.T0PS2 = 1;

    // Abilito le interruzioni del Timer0
    INTCONbits.TMR0IE = 1;

```

```

// Abilito le interruzioni del Timer0 come bassa priorità
INTCON2bits.TMR0IP = 0;

// Abilito il Timer0
T0CONbits.TMR0ON = 1;

//*****
// Abilito le interruzioni
//*****

// Abilito modalità interruzione a due livelli alta e bassa
RCONbits.IPEN = 1;

// Abilito gli interrupt ad alta priorità
INTCONbits.GIEH = 1;

// Abilito gli interrupt a bassa priorità
INTCONbits.GIEL = 1 ;

// Ciclo infinito
while (1){
}
}

```

Capisco che cominciate a rimpiangere i bei tempi in cui eravate ignari delle interruzioni e tutto funzionava...! Effettivamente il programma comincia ad essere più articolato. Questo è un caso in cui scrivere le funzioni in un altro file può aiutare a mantenere il tutto più ordinato. Nel nostro caso si è comunque scelto la via del file unico per semplificare il copia ed incolla. Si noti che per aumentare la chiarezza si è fatto uso di commenti con asterisco in maniera da evidenziare le varie sezioni del programma.

Cominciamo con la descrizione della funzione `main`. La parte iniziale non ci sorprende, visto che è identica ai nostri vecchi e cari programmi. Visto che si farà uso delle interruzioni è necessario impostare le nostre periferiche in maniera opportuna prima dei abilitarle ad interrompere.

Questa inizializzazione la si sarebbe potuta scrivere all'interno di una funzione del tipo `inizializzaPeriferiche()` da richiamare all'interno della funzione `main`. Successivamente si sarebbe potuta richiamare una funzione del tipo `abilitaInterruzioni()` in cui si abilitano appunto le interruzioni; questo renderebbe il programma di più facile lettura.

L'abilitazione dei pulsanti è piuttosto semplice e conforme ai passi base da compiere, ovvero:

```

// Abilita i resistori di pull-up sulla PORTB
EnablePullups();

// Abilito le interruzioni su PORTB
INTCONbits.RBIE = 1;

// Abilito le interruzioni su PORTB come alta priorità
INTCON2bits.RBIP = 1;

```

Nel caso del Timer0, la sua impostazione è un po' più laboriosa, visto che intervengono altri parametri per inizializzare il Timer stesso. Ciononostante i passi per l'abilitazione delle interruzioni sono sempre gli stessi.

```

// Modalità ad 16 bit
T0CONbits.T08BIT = 0;

// Clock interno
T0CONbits.T0CS = 0;

// Abilito Prescaler
T0CONbits.PSA = 0;

// Prescaler 32
T0CONbits.T0PS0 = 0;
T0CONbits.T0PS1 = 0;
T0CONbits.T0PS2 = 1;

// Abilito le interruzioni del Timer0
INTCONbits.TMR0IE = 1;

// Abilito le interruzioni del Timer0 come bassa priorità
INTCON2bits.TMR0IP = 0;

// Abilito il Timer0
T0CONbits.TMR0ON = 1;

```

I passi qui proposti possono essere anche cambiati, l'importante è che vengano comunque fatti prima dell'abilitazione del Timer. Nel nostro caso si è scelta la modalità a 16 bit in maniera da avere ritardi dell'ordine di mezzo secondo senza problemi. Sul come impostare il registro per avere dei tempi stabiliti si parlerà nel prossimo paragrafo. Si noti in particolare che per rallentare ulteriormente il conteggio si è fatto uso del prescaler, settato per rallentare il tutto dividendo per 32. In ultimo, dopo aver abilitato le interruzioni e selezionata la bassa priorità, viene abilitato il Timer, il quale inizierà il conteggio da 0. A questo punto il programma non fa null'altro che stare in un ciclo infinito in attesa delle interruzioni.

In questa applicazione si noti che sono state definite le funzioni sia per gestire le interruzioni ad alta che a bassa priorità. In particolare si è posizionata la chiamata alla funzione per la gestione delle interruzioni ad alta priorità all'indirizzo 0x08 mentre quella a bassa priorità è posta all'indirizzo 0x18. Si noti che in entrambe le funzioni la prima cosa che viene fatta è il controllo del bit di stato IF, per accertarsi che l'interruzione sia stata generata dalla periferica d'interesse. Questo è di fondamentale importanza quando sono presenti più periferiche che generano interruzioni ad un certo livello. Si noti che la periferica più importante è sempre bene metterla al primo controllo in modo da limitare il tempo di latency.

Una volta verificato che la periferica d'interesse ha generato l'interruzione, vengono svolte le operazioni di competenza. Alla fine del blocco `if` che racchiude le istruzioni da eseguire, viene resettato il bit IF della periferica, in maniera da permettere il corretto riconoscimento di nuovi interrupt sia della periferica che di altre.

Eseguendo il programma sulla scheda di sviluppo Freedom II, opportunamente impostata con i Jumper come nel caso del programma `Hello_World`, si può vedere che il lampeggio ed il riconoscimento della pressione del pulsante avvengono in maniera fluida. Inoltre il programma per come è organizzato potrebbe facilmente gestire altre interruzioni senza compromettere la fluidità del programma originale.

Si fa notare che se si include il file di libreria `timers.h` che viene installato con il compilatore C18 è possibile limitare il numero d'istruzioni del programma. Ciononostante la compilazione non è realmente ottimizzata visto che dietro la chiamata delle funzioni di libreria sono comunque presenti i passi descritti nell'esempio. Per maggiori informazioni sulla

---

libreria `timers.h` si rimanda alla documentazione ufficiale che è possibile trovare nella cartella `doc` presente nel cartella d'installazione del compilatore. Un esempio di utilizzo della libreria verrà comunque presentato negli esempi presentati nei prossimi Capitoli.

## Impostare il tempo del Timer

Come visto nell'esempio precedente si è potuto far lampeggiare il LED 0 con tempi stabiliti dal `Timer0`. Vediamo di capire qual'era il tempo che caratterizzava il lampeggio.

Sappiamo che sulla scheda Freedom II è presente un quarzo da 20MHz. Questo significa che il periodo del clock utilizzato è pari a  $0.05\mu s$  ovvero:

$$T_{OSC} = \frac{1}{F_{OSC}} = \frac{1}{(20 \cdot 10^6)} = 0.05 \mu s$$

Si ricorda in particolare che il tempo di esecuzione di una istruzione è pari a quattro cicli di clock dunque il clock che va ad incrementare il nostro `Timer` è pari a  $0.2\mu s$ , ovvero:

$$T_{TIMER0} = T_{OSC} \cdot 4 = 0.05 \cdot 4 = 0.2\mu s$$

Nel nostro esempio il periodo di  $0.2\mu s$  viene in realtà rallentato dal prescaler che è stato impostato a 32. Questo significa che il periodo del clock uscente dal prescaler sarà 32 volte maggiore:

$$T_{PRESCALER} = T_{OSC} \cdot 4 \cdot 32 = 0.05 \cdot 128 = 6.4\mu s$$

Il clock in uscita dal prescaler è effettivamente quello utilizzato per far incrementare il nostro `Timer0`, il quale è stato impostato per lavorare in modalità a 16 bit. Questo significa che conterà da 0 a 65535. Quando avverrà l'overflow, ovvero il passaggio da 65535 a 0, viene generato il nostro interrupt che fa cambiare lo stato al LED 0, quindi il nostro LED viene cambiato di stato ogni 0.42 secondi<sup>106</sup>.

*Cosa bisogna fare per ottenere 0.5 secondi?*

Un modo di procedere è il seguente. Considerando i tempi elevati bisognerà sicuramente far uso del prescaler, che come detto ritarderà il clock ottenendo un periodo di  $6.4\mu s$  che come abbiamo appena visto non è sufficiente per ottenere il tempo desiderato neanche facendo contare fino al massimo il `Timer0` impostato in modalità a 16 bit. Questo significa che bisognerà utilizzare il prescaler dividendo per 64 piuttosto che per 32, ottenendo in questo modo :

$$T_{PRESCALER} = T_{OSC} \cdot 4 \cdot 64 = 0.05 \cdot 128 = 12.8\mu s$$

In questo caso moltiplicando il nuovo periodo per 65536 (si è considerato anche lo 0) si ha che il nostro ritardo totale è 0.839s, ovvero il doppio di prima! Questa volta il nostro `Timer0` rallenta troppo. Per ottenere esattamente 0.5s, quello che si fa è non far partire il `Timer0` da 0, ovvero si carica un valore nel timer in modo che raggiunga l'overflow in minor tempo. Questo significa che potremo in realtà ottenere il nostro tempo...

---

<sup>106</sup> Nel conteggio del nostro tempo si e sono trascurati i due cicli di clock di ritardo richiesti per la sincronizzazione del clock.

---

Per capire il valore da caricare nel Timer0 si divide il tempo che si vuole ottenere per il periodo del clock ottenuto in uscita dal prescaler, ovvero:

$$CONTEGGI = \frac{0.5}{T_{PRESCALER}} = \frac{0.5}{0.0000128} = 39062.5$$

Il numero di conteggi non è il valore da caricare nel Timer0, bensì il numero di conteggi che deve fare il Timer0 prima di andare in overflow. Questo significa che il numero da caricare nel Timer0 è pari a:

$$TIMER0 = (2^{16} + 1) - 39062.5 = 65536 - 39062.5 = 26473.5$$

Dal momento che nel Timer0 si possono caricare solo numeri interi sarà necessario approssimare o troncare il numero, in questo caso dunque il numero da caricare è 26473. Va bene troncare il numero poiché questo non sarebbe comunque preciso. Come precedentemente detto per avere tempi precisi sarebbe necessario anche avere un oscillatore preciso.

Qualche altra nota è importante. Il valore 26473 deve essere caricato all'interno di due registri ovvero TMR0L e TMR0H. Questo significa che bisogna spezzare il valore 26473 in due byte, per fare questo si deve convertire il numero in binario (per esempio usando la calcolatrice di Windows). Il numero binario è: 110011101101001. Tale numero deve esser poi diviso in due byte:

01100111 - 01101001

Si noti che la divisione in due gruppi è stata fatta partendo da destra. Il primo byte di destra dovrà essere caricato nel registro TMR0L, mentre l'altro deve essere caricato nel registro TMR0H, ovvero:

```
TMR0H = 0b01100111;  
TMR0L = 0b01101001;
```

Si osservi che il registro TMR0H è stato scritto per primo. Questo passo è obbligatorio poiché il registro TMR0H non rappresenta effettivamente gli otto bit più significativi del Timer0 in modalità 16 bit, bensì un buffer (registro di supporto), il cui valore viene caricato nel registro TMR0H quando avviene la scrittura del registro TMR0L. Questo significa che per caricare un valore in modalità a 16 bit è necessario caricare il valore in TMR0H prima di scrivere in TMR0L. Per maggiori dettagli si faccia riferimento al datasheet del PIC utilizzato.

Le istruzioni sopra citate per caricare il valore nel Timer0 devono essere eseguite subito dopo aver attivato il Timer0 dunque subito dopo l'inizializzazione del Timer0. Oltre a questo bisogna ricaricare tale valore ogni volta che si entra nella funzione d'interrupt del Timer0<sup>107</sup>.

Dal momento che il Timer0 può essere gestito anche in modalità ad 8 bit si può anche prendere in considerazione il fatto di far lavorare il registro in tale modalità. Il tutto dipende dai tempi che dobbiamo ottenere. Nel nostro caso per esempio, pur mettendo il prescaler a 256 si sarebbe potuto ottenere un ritardo massimo di circa 13ms.

---

<sup>107</sup> Si noti che il tempo da caricare nella funzione d'interrupt dovrebbe in realtà considerare anche il tempo di latenza per l'avvio della routine di gestione e il tempo di esecuzione per caricare il valore stesso. Si noti che il tempo di latenza potrebbe non essere trascurabile se la direttiva `#pragma` viene dichiarata con l'opzione `save`, ovvero con l'opzione di salvataggio di un blocco di memoria.

---

In ultimo si fa notare che qualora si debbano ottenere ritardi ben superiori al secondo, per esempio 1 minuto, ovvero in casi in cui la combinazione del Timer e il prescaler non sono sufficienti per ottenere il ritardo voluto; quello che si fa è inserire un ulteriore conteggio all'interno della procedura d'interrupt<sup>108</sup>. Se il tempo dovesse raggiungere valori dell'ora la soluzione proposta va ancora bene, ma si potrebbe anche prendere in considerazione l'utilizzo di un Clock Calendar esterno o ottenuto per mezzo del Timer1 ed un quarzo da 32768KHz.

---

<sup>108</sup> Un'alternativa potrebbe essere anche quella di diminuire la frequenza del Clock principale, ma questo non è sempre possibile, visto che alcune periferiche richiedono delle frequenze minime di funzionamento, si pensi per esempio al modulo USB, I2C e USART.

# Capitolo IX

## Utilizziamo il modulo PWM interno al PIC

L'esperienza ora raggiunta ci permette di fare un altro passo in avanti. In questo Capitolo viene introdotto il modulo PWM per mezzo del quale è possibile controllare la velocità di motori, la luminosità di lampade o anche far parlare un microcontrollore! In un mondo che richiede sempre più energia, la tecnica del PWM permette di ottenere ottime efficienze nel controllo e distribuzione di energia ai dispositivi controllati. In questo Capitolo verrà preso in considerazione il PIC18F4550.

### Descrizione dell'hardware e sue applicazioni

Un segnale PWM (Pulse Width Modulation) rappresenta un segnale digitale identificato da un livello alto e basso. L'ampiezza che esso raggiunge è di poco interesse e può variare da applicazione ad applicazione. Quello che realmente identifica un segnale PWM è la sua frequenza e duty cycle. La frequenza è definita come l'inverso del suo periodo, ovvero il tempo che intercorre tra il ripetersi di un ciclo.

$$f = \frac{1}{T}$$

Normalmente le frequenze utilizzate sono dell'ordine di pochi KHz fino a centinaia di KHz. In particolari applicazioni vengono utilizzate anche frequenze dell'ordine del MHz. Il duty cycle quantifica invece l'ampiezza dell'impulso in rapporto al periodo, quindi:

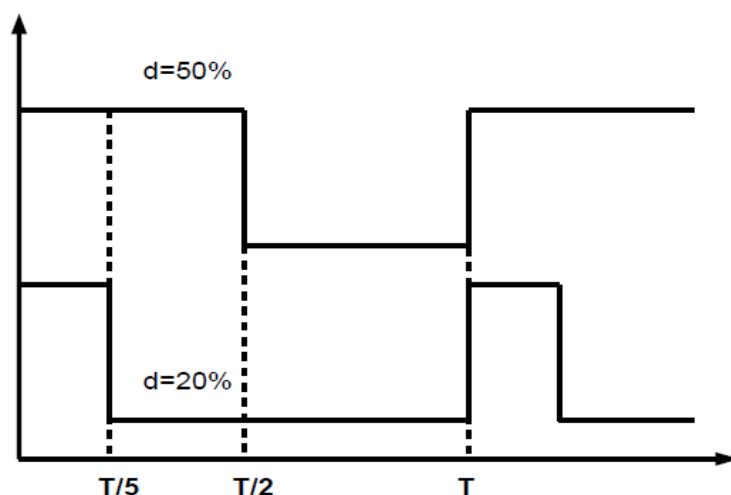
$$d = \frac{T_{ON}}{T}$$

Dove  $T_{ON}$  rappresenta il tempo che l'impulso rimane a livello alto. Si capisce che  $T_{ON}$  risulterà sempre minore o uguale al periodo  $T$ . Frequentemente il duty cycle viene espresso in percentuale:

$$d = \frac{T_{ON}}{T} \cdot 100$$

In Figura 55 è possibile vedere diversi impulsi di ugual periodo ma con diverso duty cycle. Nel caso in cui il segnale PWM venga generato per mezzo di un microcontrollore, un altro parametro caratteristico è rappresentato dal numero di bit con il quale viene generato. Se per esempio il numero di bit è 8, il periodo  $T$  verrà suddiviso in 256 intervalli. La durata di un intervallo è calcolata facendo il rapporto tra il periodo  $T$  e il numero d'intervalli. Nel caso particolare del PIC18F4550 si ha che il modulo PWM ha una risoluzione di 10 bit ovvero si hanno 1024 intervalli di tempo per il quale il nostro periodo può essere diviso. A seconda del periodo utilizzato si avrà che l'intervallo temporale base avrà una durata differente.

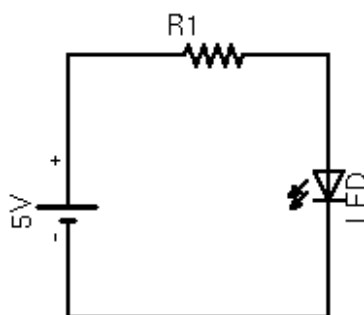




**Figura 55:** Due segnali PWM a pari frequenza ma duty cycle differente

Facciamo ora un passo indietro e ripensiamo a cosa significa PWM cioè Pulse Width Modulation ovvero modulazione della larghezza dell'impulso.

Un segnale che venga modulato ovvero cambiato, è modulato in PWM quando ha un'ampiezza a due livelli, per questo lo abbiamo definito digitale, una frequenza fissa, ed un duty cycle variabile. Modulare il segnale in PWM significa appunto variare la larghezza del nostro impulso ovvero il duty cycle. Vediamo ora un semplice esempio che mette in evidenza una delle ragioni per cui si dice che il PWM è efficiente ovvero permette di risparmiare energia. Si consideri la Figura 56, di un LED con in serie un resistore.



**Figura 56:** PWM Schema elettrico di un normale controllo di un LED

Questo schema è normalmente usato in molte applicazioni, ma posso dire che questo non è il modo con cui i LED di retroilluminazione dello schermo del vostro cellulare sono realmente alimentati. La ragione è che la resistenza che viene posta per limitare la corrente dissipa molta energia. Considerando per esempio un LED con una caduta di potenziale pari ad 1.6V e sul quale si voglia far scorrere una corrente di 10mA, alimentando il tutto con 5V, si ha che la resistenza serie che deve essere utilizzata è<sup>109</sup>:

$$R = \frac{(V_{cc} - V_{LED})}{I_{LED}} = \frac{(5 - 1.6)}{0.01} = 340 \text{ ohm}$$

<sup>109</sup> Il valore ottenuto non rientra nel valore standard della serie E12. Il valore più vicino è 330 ohm. Questo valore farà però scorrere una corrente lievemente maggiore pari a 10.3mA.

---

Si capisce dunque che la potenza dissipata dal resistore, per il solo compito di limitare la corrente del diodo LED è pari a:

$$P_{LED} = V_R \cdot I_{LED} = (V_{CC} - V_{LED}) \cdot I_{LED} = 3.4 \cdot 0.01 = 0.034 \text{ W}$$

Tale potenza anche se apparentemente irrisoria, è oro colato qualora si stia alimentando un dispositivo con una batteria delle dimensioni di un cucchiaino!

Il PWM ci viene in aiuto grazie al fatto che modulando il segnale ovvero accendendo e spegnendo il LED è possibile controllare l'energia che viene trasferita. L'energia viene a dipendere dall'area del nostro segnale quando è posto ad 1. Si ricorda che la potenza rappresenta l'energia consumata da un dispositivo nell'unità di tempo. Dunque se con un segnale PWM riusciamo a controllare l'energia che trasferiamo ad un dispositivo, quello che stiamo facendo è controllare la potenza di quest'ultimo! Questo significa che controllando in maniera opportuna il duty cycle del nostro segnale PWM è possibile controllare un LED senza aver bisogno del resistore, ovvero si risparmierebbe la potenza dissipata sul resistore stesso. Il duty cycle deve essere opportunamente controllato, poiché se eccede i limiti che un determinato dispositivo riesce a sopportare, porterebbe alla rottura di quest'ultimo.

Nonostante non sia presente la resistenza per limitare la corrente è comunque necessario che il LED sia inserito in un circuito con filtro passa basso al fine di garantire che la tensione non ecceda i limiti del diodo stesso. Normalmente i driver per LED comandati in PWM non sono altro che convertitori DC-DC di tipo switching (normalmente in modalità buck o boost), ed il filtro passa basso è rappresentato dall'induttore e la resistenza di feedback<sup>110</sup> e/o Ron dei MOS utilizzati per il controllo dei LED stessi.

La tecnica del PWM viene utilizzata in molte altre applicazioni, si pensi per esempio agli alimentatori switching con cui si ricaricano i cellulari moderni, avrete notato che sono divenuti più leggeri che in passato! Larga applicazione è anche fatta in dispositivi alimentati a batteria grazie alle alte efficienze dei regolatori switching rispetto ai vecchi regolatori lineari<sup>111</sup>. Per mezzo della modulazione PWM è anche possibile realizzare sistemi di comunicazione tra due sistemi, alla pari di altre tecniche di trasmissione o modulazione. La modulazione PWM trova ampio utilizzo anche nei sistemi di controllo per motori, o sistemi di potenza in generale, in cui bisogna controllare la potenza di un carico. Nel caso di motori si traduce nel fatto che grazie al PWM è possibile controllare la loro velocità o coppia.

In ultimo è bene ricordare che la tecnica PWM permette di realizzare un convertitore DAC (Digital to Analog Converter) permettendo per esempio di far parlare un PIC, convertendo file .wav.

I PIC18 possiedono tutti al loro interno almeno un modulatore PWM, definito modulo CCP. Frequentemente possiedono anche due moduli CPP e ECCP ovvero un modulo CPP migliorato. Negli esempi che seguiranno si prenderà in considerazione il PIC18F4550 che possiede al suo interno due moduli CCP standard con l'opzione di abilitare anche un modulo ECCP. Il modulo ECCP non verrà discusso, ma è comunque bene menzionare il fatto che tale

---

<sup>110</sup> I LED sono spesso nominati componenti in corrente poiché vengono normalmente pilotati in corrente piuttosto che in tensione. Il controllo della corrente che attraversa il LED, ovvero il controllo del duty cycle del segnale PWM, è effettuato grazie alla resistenza di feedback collegata in serie con il LED stesso. Il valore della resistenza di feedback è dell'ordine del ohm, dunque la potenza dissipata dal resistore è esigua se confrontata al caso in cui la resistenza venga usata per limitare la corrente.

<sup>111</sup> Gli alimentatori lineari quali la serie 78, hanno ancora ampio utilizzo grazie al basso costo e semplicità di utilizzo. In applicazioni di precisione i regolatori lineari, siano LDO (Low Drop Out) o meno, hanno comunque la meglio rispetto agli alimentatori switching visto che generano meno rumore di quest'ultimi.

---

modulo è ottimizzato per controllare strutture Half Bridge e Full Bridge, spesso utilizzate per il controllo di motori. I moduli CCP possono anche essere utilizzati in modalità Capture e modalità Compare; queste due modalità non verranno discusse. Vi basti sapere che possono essere utilizzate per sincronizzare eventuali eventi esterni od interni, tra loro, facendo un conteggio sugli stessi. Per maggiori informazioni si rimanda al datasheet del PIC utilizzato.

Da quanto abbiamo studiato sui microcontrollori e sulla modulazione PWM, si sarà anche giunti alla conclusione che un microcontrollore potrebbe realizzare un segnale PWM semplicemente controllando un qualunque pin del microcontrollore stesso. Effettivamente questo è possibile, però per ottenere la stessa risoluzione offerta dal modulo PWM del PIC, ovvero 10bit, il software per il controllo richiederebbe un utilizzo elevato della CPU, la quale non riuscirebbe più a compiere altre mansioni. Per mezzo del modulo CCP è invece possibile con poche impostazioni impostare il modulo e avviarlo. Questo lavorerà indipendentemente senza ulteriore costo da parte dell'utilizzo della CPU. Questa dovrà intervenire solo per cambiare il duty cycle quando richiesto.

Per ulteriori informazioni sulla tecnica PWM si rimanda al Tutorial “PWM, Pulse Width Modulation” che è possibile scaricare dal sito [www.LaurTec.com](http://www.LaurTec.com).

## I registri interni per il controllo del PWM

La microchip fornisce una semplice libreria per mezzo della quale è possibile attivare, disattivare e controllare il modulo PWM. Per tale ragione i dettagli sui singoli registri verrà evitata. In ogni modo si capisce che sarà presente un registro con i bit necessari per selezionare la modalità PWM e l'opzione di attivare il modulo o meno. Un registro per selezionare il duty cycle ed un altro per impostare la frequenza o periodo del PWM.

Per poter utilizzare la libreria Microchip bisogna includere il relativo file di libreria nel seguente modo `#include <pwm.h>`. A seconda del modello del PIC di cui si sta facendo uso, possono essere presenti fino a 5 periferiche per il PWM<sup>112</sup>. Per poter distinguere le varie periferiche PWM ogni funzione deve essere terminata con il numero della periferica PWM a cui si sta facendo riferimento. Il PIC18F4550 possiede due moduli CCP dunque le funzioni per il suo controllo termineranno per 1 o 2 a seconda che si utilizzi il modulo CCP1 o CCP2. Il modulo CCP1 possiede l'uscita al pin RC2 mentre il modulo CCP2 possiede l'uscita multiplexata con i pin RC1 ed RB3. Di default il pin assegnato è RC1, normalmente utilizzato anche in altri PIC con due moduli CCP. Per cambiare l'uscita del modulo CCP2 ad RB3 bisogna cambiare il valore del Configuration Register, ovvero per mezzo della direttiva `#pragma` bisogna impostare il valore opportuno<sup>113</sup>:

```
// CPP2 è sul pin RB3
#pragma config CCP2MX = OFF
```

oppure:

```
// CPP2 è sul pin RC1
#pragma config CCP2MX = ON
```

---

<sup>112</sup> La libreria supporta le varie versioni del modulo PWM relative ai vari PIC. Di questo non si è parlato ma è bello pensare che la libreria penserà a tutto!

<sup>113</sup> Si ricorda che i valori di configurazione di ogni PIC sono riportati nel file `hlpPIC18ConfigSet.chm` che è possibile trovare nella directory doc in cui è stato installato il C18.

Le funzioni dedicate per il controllo PWM, in particolare al modulo standard CCP, presenti all'interno della libreria pwm.h sono riportate in Tabella 8. Per il modulo ECCP sono presenti anche altre funzioni; queste non sono però discusse in questa sede.

Funzioni	Descrizione
void <b>ClosePWMx</b> (void)	Disabilita il canale x PWM
void <b>OpenPWMx</b> (char)	Apri il canale x PWM
void <b>SetDCPWMx</b> (unsigned int)	Imposta un nuovo duty cycle per il canale PWM

**Tabella 8:** Funzioni di controllo del modulo CPP standard

#### **void ClosePWMx (void)**

Per mezzo di questa funzione è possibile chiudere il canale PWM d'interesse cambiando la x con il numero del canale che si desidera controllare.

#### **Parametri:**

void.

#### **Ritorna:**

void.

#### **Esempio:**

```
// Chiude il modulo 1
ClosePWM1 ();
```

#### **void OpenPWMx (char period)**

Per mezzo di questa funzione è possibile impostare il periodo del segnale PWM. Si ricorda che il periodo è l'inverso della frequenza  $f = 1/T$ . Il periodo da inserire non è in realtà il periodo del segnale PWM ma è ad esso correlato.

#### **Parametri:**

**period** : La formula che lega *period* al periodo attuale è:

$$Periodo\ PWM = [(period) + 1] \cdot 4 \cdot T_{OSC} \cdot TMR2_{prescaler}$$

Da questa relazione si capisce che per poter utilizzare il segnale PWM bisogna anche aprire il timer TMR2. Infatti il periodo del PWM viene a dipendere dal valore del Prescaler del timer TMR2. Un altro parametro che interviene nel calcolo del periodo del segnale PWM è il periodo del Clock generato dal nostro quarzo. Per calcolare questo basta fare l'inverso della frequenza del quarzo stesso, qualora questo sia generato per mezzo di un quarzo esterno, si ha  $T_{OSC} = 1/F_{OSC}$ . Vediamo la formula inversa per il calcolo della variabile *period* una volta note le altre grandezze:

---


$$period = \frac{Periodo\ PWM}{4 \cdot T_{OSC} \cdot TMR2_{prescaler}} - 1$$

che può anche essere riscritta nel seguente modo:

$$period = \frac{Periodo\ PWM \cdot F_{OSC}}{4 \cdot TMR2_{prescaler}} - 1$$

Per poter controllare il timer TMR2 si può far uso della libreria C18 timers.h, piuttosto che controllare i singoli registri come fatto negli esempi di controllo del Timer0.

#### Ritorna:

void.

#### Esempio:

Si veda il Progetto come esempio.

---

#### void SetDCPWMx (unsigned int *dutycycle*)

Per mezzo di questa funzione è possibile impostare il duty cycle del segnale PWM.

#### Parametri:

**dutycycle** : Il duty cycle può variare da un minimo di 0 a un massimo 1023 (10bit). Un duty cycle pari a 0 vincola il segnale PWM a 0 mentre un duty cycle pari a 1023 vincola il segnale PWM a 1.

#### Ritorna:

void

#### Esempio:

```
// Aggiorno il duty cycle
SetDCPWM2 (820);
```

---

Per maggiori informazioni sul modulo PWM si rimanda al datasheet del PIC utilizzato. Si raccomanda inoltre la lettura del paragrafo relativo ai limiti operativi.

## Esempio di utilizzo del modulo PWM

Vediamo ora un semplice esempio di utilizzo della tecnica PWM per controllare l'intensità luminosa di un LED. Facendo uso della scheda di sviluppo Freedom II piuttosto che collegare hardware esterno, si controllerà l'intensità luminosa del LED di retroilluminazione utilizzato nel display LCD. Tale controllo potrebbe risultare particolarmente utile se associato

---

all'intensità luminosa dell'ambiente circostante, la quale può essere rilevata per mezzo della fotoresistenza presente sempre sulla scheda Freedom II. Questa tecnica viene spesso utilizzata su PC e cellulari per ridurre la luminosità del display a seconda della luminosità dell'ambiente esterno, permettendo di risparmiare energia. Vediamo il nostro esempio:

```
#include <p18f4550.h>
#include <pwm.h>
#include <timers.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF
#pragma config CCP2MX = ON

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici
//CPP2MX = ON        Il modulo CCP è posto su RC1

void main (void){

    // Variabile usata per creare un conteggio fittizio di pausa
    int i;

    // Periodo del segnale PWM
    unsigned char period;

    // Duty Cycle
    int duty_cycle = 0;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi ed RC1 come uscita
    LATC = 0x00;
    TRISC = 0b11111101;

    // Imposto PORTD tutti ingressi
    LATD = 0x00;
    TRISD = 0xFF;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Apro il TMR2 per il PWM
    OpenTimer2 (TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);

    // Imposto una frequenza di 20KHz
    period = 249;

    // Apro il moldulo PWM
```

```

OpenPWM2(period);

// Ciclo infinito
while (1) {

    // Aggiorno il dutycycle
    SetDCPWM2 (duty_cycle);

    // Incremento il dutycycle
    duty_cycle++;

    // Controllo che non sia maggiore di 2^10
    if (duty_cycle > 1023) {
        duty_cycle =0;
    }

    // Pausa
    for (i=0; i<1000; i++) {

    }

}
}

```

Come prima cosa, si noti che diversamente dagli altri programmi si sono incluse le seguenti librerie:

```

#include <pwm.h>
#include <timers.h>

```

Maggiori informazioni sulle librerie possono essere trovate all'interno della directory doc presente nella directory d'installazione del C18. Pur essendo RC1 il bit di default per il modulo CCP2 si è preferito esplicitarlo per mezzo della direttiva:

```

#pragma config CCP2MX = ON

```

Per quanto riguarda le impostazioni dei pin si noti che il pin RC1 è stato impostato come uscita, in modo da permettere il corretto funzionamento del modulo PWM.

```

// Imposto PORTC tutti ingressi ed RC1 come uscita
LATC = 0x00;
TRISC = 0b11111101;

```

Come passo successivo si è abilitato il modulo Timer2 per poter impostare il Prescaler utilizzato dal modulo CCP2.

```

// Apro il TMR2 per il PWM
OpenTimer2( TIMER_INT_OFF & T2_PS_1_1 & T2_POST_1_1);

```

Si noti che il il Prescaler è impostato ad 1:1 come anche il Postscaler. Questo significa che ai fini del PWM è in realtà ininfluenza, perlomeno nel nostro caso. Infatti nella formula del calcolo del periodo va a moltiplicare per 1.

Dopo aver impostato il Timer2, si imposta il periodo del modulo PWM e lo si attiva.

```

period = 249;

```

```
// Apro il modulo PWM
OpenPWM2(period);
```

Si noti che dal momento in cui il modulo utilizzato è CCP2, la funzione chiamata è OpenPWM2. Qualora si utilizzasse il modulo CCP1 con uscita al pin RC2, si sarebbe dovuta richiamare la funzione OpenPWM1. Inserendo il nostro periodo pari a 249 all'interno della formula:

$$Periodo\ PWM = [(period) + 1] \cdot 4 \cdot T_{OSC} \cdot TMR2_{prescaler}$$

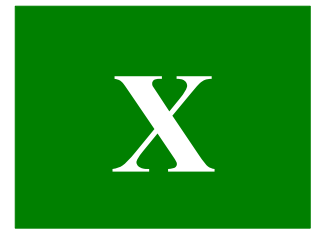
si ha che il periodo è pari a 0.05ms ovvero pari ad una frequenza di 20KHz.

Una volta impostato il periodo e avviato il modulo PWM si effettua la modulazione del segnale all'interno del ciclo infinito. La modulazione consiste semplicemente nell'incrementare il duty cycle e aggiornare il suo valore nel modulo PWM. In questo modo si ha che l'intensità del LED di retroilluminazione tenderà ad aumentare d'intensità fino a raggiungere il valore massimo. Per rendere la variazione del LED più lenta, si è posto un semplice ciclo di ritardo. All'interno del nostro ciclo è anche presente il seguente controllo:

```
// Controllo che non sia maggiore di 2^10
if (duty_cycle > 1023) {
    duty_cycle = 0;
}
```

Tale controllo è necessario dal momento che il modulo PWM del PIC ha una risoluzione a 10 bit. Raggiunto il valore massimo, il duty cycle viene posto nuovamente a 0 per iniziare un nuovo conteggio. Da un punto di vista visivo si ha che il LED di retroilluminazione viene spento.





# Capitolo X

## Utilizziamo un display alfanumerico LCD

In questo Capitolo viene introdotta una periferica esterna al PIC, ovvero indipendente dal PIC. Il suo utilizzo è però così frequente nonché importante che è bene parlarne. I display LCD alfanumerici permettono infatti con pochi soldi di creare un'interfaccia grafica professionale, permettendo ai nostri programmi di colloquiare con l'utilizzatore finale. In particolare al fine di mantenere la discussione quanto più semplice possibile, l'utilizzo del display viene ricondotto all'utilizzo della libreria per controllarlo. In questo modo con poche righe saluteremo nuovamente il mondo...

### Descrizione dell'hardware e sue applicazioni

I display alfanumerici LCD sono ormai molto popolari e permettono con modica spesa di aggiungere un pizzico di professionalità ad ogni circuito. Per mezzo di tali display è inoltre possibile realizzare un'ottima interfaccia macchina utente grazie ai menù che è possibile scrivere direttamente sul display. In commercio sono presenti molti tipi di display alfanumerici LCD di varie dimensioni, quelle più note ed utilizzate sono 8x1, 8x2, 16x1, 16x2, 20x2, 40x4, dove il primo numero indica il numero dei caratteri<sup>114</sup> che è possibile scrivere su ogni riga mentre il secondo rappresenta il numero di righe disponibili. Ogni LCD possiede almeno un controllore che permette la comunicazione tra il microcontrollore e il display LCD. Sono presenti diversi tipi di controllori con diversi tipi set di istruzioni necessarie per la comunicazione col controllore stesso. Il più utilizzato è senza dubbio il controllore HD44780 della Hitachi. Sono presenti anche altre sigle d'integrati realizzati da altre case costruttrici ma che sono compatibili con questo controllore; infatti tale controllore è divenuto uno degli standard per il controllo dei display alfanumerici.

Il controllore HD44780, possiede una memoria interna in cui sono memorizzati i caratteri che possono essere scritti sul display, il codice assegnato ad ogni lettera è quello ASCII, dunque scrivendo un carattere o un intero sul display questo verrà interpretato secondo la tabella del codice ASCII. Oltre al set di caratteri standard si ha anche la possibilità di creare dei caratteri personali e scriverli all'interno della memoria del controllore. I caratteri scritti possono poi essere richiamati per la loro visualizzazione. Per ulteriori informazioni sulle caratteristiche e funzionalità del controllore HD44780 si rimanda al relativo datasheet.

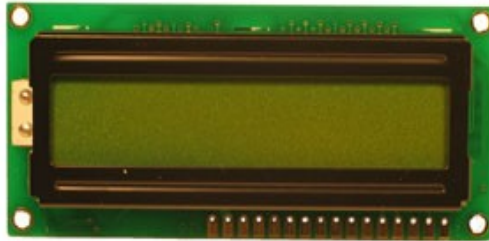
Ogni display LCD possiede varie linee di controllo in particolare un bus per la trasmissione dei dati composto da 8 linee, una linea di Enable<sup>115</sup>, una linea R/W per scrivere o leggere dal/sul controllore e una linea RS per distinguere l'invio di un comando da un carattere. Oltre a queste linee, necessarie per il controllo del display, è presente il pin per il contrasto. Il controllore, al fine di risparmiare pin sul PIC può essere utilizzato in modalità 4 bit piuttosto che a 8 bit, ovvero si fa uso di sole 4 linee dati. La piedinatura dei display è

<sup>114</sup> Ogni carattere è contenuto all'interno di una piccola matrice di punti per mezzo dei quali si ottiene la forma del carattere stesso o anche di un simbolo.

<sup>115</sup> Le linee di Enable possono anche essere due ma per il 16x2, 16x1 si ha una sola linea di enable.

---

generalmente standard ma potrebbe variare da quella sotto riportata, in particolare i pad delle varie linee potrebbero essere localizzate in alto a sinistra (come Freedom II) piuttosto che in basso a destra, l'ordine dei pin rimane in genere invariato. In Figura 57 è riportato un esempio di Display LCD 16x2.



**Figura 57:** Esempio di Display LCD 16x2 con retroilluminazione

pin 1 = GND (il pin 1 è generalmente indicato sul display stesso)  
pin 2 = Vcc (+5V)  
pin 3 = Contrasto  
pin 4 = RS  
pin 5 = R/W (collegato a massa in applicazioni in cui non si legge dal controllore)  
pin 6 = E  
pin 7 = DB0 (non usato in modalità 4 bit)  
pin 8 = DB1 (non usato in modalità 4 bit)  
pin 9 = DB2 (non usato in modalità 4 bit)  
pin 10 = DB3 (non usato in modalità 4 bit)  
pin 11 = DB4 (D0 in modalità 4 bit)  
pin 12 = DB5 (D1 in modalità 4 bit)  
pin 13 = DB6 (D2 in modalità 4 bit)  
pin 14 = DB7 (D3 in modalità 4 bit)  
pin 15 = LED+  
pin 16 = LED-

Il C18 possiede una libreria dedicata per il controllo dei display LCD ma per ragioni di semplicità si parlerà della libreria che ho personalmente realizzato<sup>116</sup> ovvero la libreria LCD\_44780.h. La libreria LCD\_44780 è preimpostata per lavorare correttamente con la scheda Freedom II, ma con pochi semplici passi, può essere adattata per funzionare con qualunque applicazione 4 bit. Le linee di cui si è parlato sopra possono essere infatti collegate ad un qualunque pin del PIC lasciando ampia libertà. Naturalmente per semplicità è sempre bene assegnare i pin con un criterio logico ma non è obbligatorio. Freedom II possiede un LCD 16x2 con retroilluminazione e possiede anche il trimmer per il contrasto. I pin di controllo sono collegati sulla PORTD.

---

<sup>116</sup> Il file di libreria è disponibile al sito [www.LaurTec.com](http://www.LaurTec.com).

---

## Utilizzo della libreria LCD

Come detto, la libreria di cui si parlerà è la libreria LCD\_44780, per il suo corretto utilizzo bisogna includere il file LCD\_44780.h ed il file LCD\_44780.lib. Dal momento che la libreria fa uso della libreria delay.h, è anche necessario includere il file delay.lib<sup>117</sup>. Bisogna inoltre aggiornare i percorsi di libreria per poter permettere al compilatore di trovare i file d'interesse. Come detto la libreria è già ottimizzata per lavorare correttamente su Freedom II, ciononostante può essere facilmente adeguata ad altre esigenze. Dal momento che la libreria è piuttosto flessibile, a scopo precauzionale, ho previsto la generazione di alcune warning nel caso si utilizzino le impostazioni standard. Se infatti si include solo il file:

```
#include <LCD_44780.h>
```

Compilando i nostri programmi si avranno i seguenti messaggi di warning:

```
Warning [2105] LCD_D0 has been not defined, LATDbits.LATD4 will be used
Warning [2105] LCD_D1 has been not defined, LATDbits.LATD5 will be used
Warning [2105] LCD_D2 has been not defined, LATDbits.LATD6 will be used
Warning [2105] LCD_D3 has been not defined, LATDbits.LATD7 will be used
Warning [2105] LCD_RS has been not defined, LATDbits.LATD2 will be used
Warning [2105] LCD_E has been not defined, LATDbits.LATD3 will be used
Warning [2105] LCD_RW has been not defined, LATDbits.LATD1 will be used
Warning [2105] LCD_LED has been not defined, LATCbits.LATC1 will be used
```

Questi messaggi stanno ad indicare che le varie variabili interne non sono state impostate e quelle di Default verranno utilizzate. In particolare quelle di Default rappresentato proprio quelle per poter lavorare correttamente con Freedom II.

### Nota:

L'utilizzo d'impostazioni non idonee al proprio progetto o scheda di sviluppo potrebbe danneggiare il PIC o altro hardware.

Come detto è bene sempre compilare un programma senza alcun messaggio di warning, dunque la libreria prevede un modo per non far visualizzare tali messaggi. Il modo consiste nel comunicare alla libreria l'intenzione di utilizzare i valori di Default. Per fare questo è necessario definire il nome LCD\_DEFAULT prima della chiamata della libreria, ovvero:

```
#define LCD_DEFAULT
#include <LCD_44780.h>
```

In questo modo i controlli interni della libreria sapranno che l'utente è consapevole che i valori utilizzati dalla libreria sono quelli di Default.

Oltre alla libreria è necessario impostare, per mezzo dei registri TRISx del PIC, i vari pin utilizzati dalla libreria stessa in modo opportuno. In particolare i pin utilizzati dalla libreria sono:

---

<sup>117</sup> Il file delay.h non è necessario che venga incluso visto che viene incluso dalla libreria LCD\_44780.h. In ogni modo è necessario che i percorsi del compilatore siano propriamente impostati per permettere al compilatore di trovare il file header.

**RD1:** Deve essere impostato come Output. Linea RW.  
**RD2:** Deve essere impostato come Output. Linea RS.  
**RD3:** Deve essere impostato come Output. Linea E.  
**RD4:** Deve essere impostato come Output. Linea D4.  
**RD5:** Deve essere impostato come Output. Linea D5.  
**RD6:** Deve essere impostato come Output. Linea D6.  
**RD7:** Deve essere impostato come Output. Linea D7.

**RC1:** Deve essere impostato come Output. Controllo LED retroilluminazione.

Dunque TRISD deve essere impostato caricando il valore 0000000x dove x è indifferente e in particolare può variare a seconda dell'applicazione. Se non usato è bene impostarlo come Input ovvero ad 1. Il registro TRISC deve essere impostato con il valore xxxxxx0x dove le x dovranno essere impostate a seconda dell'applicazione. In particolare se gli altri pin non sono utilizzati è bene impostarli come input, ovvero ad 1.

Per poter propriamente utilizzare il Display LCD di Freedom II è necessario abilitarlo per mezzo dei Jumper. La configurazione che verrà utilizzata negli esempi che seguiranno è riportata in Figura 58.

ON	OFF	JUMPER
<input checked="" type="checkbox"/>	<input type="checkbox"/>	USB_CP
<input checked="" type="checkbox"/>	<input type="checkbox"/>	USB_DET
<input type="checkbox"/>	<input checked="" type="checkbox"/>	SCL
<input type="checkbox"/>	<input checked="" type="checkbox"/>	SDA
<input checked="" type="checkbox"/>	<input type="checkbox"/>	ANALOG
<input type="checkbox"/>	<input checked="" type="checkbox"/>	INT
<input checked="" type="checkbox"/>	<input type="checkbox"/>	TEMP
<input checked="" type="checkbox"/>	<input type="checkbox"/>	SPK
<input type="checkbox"/>	<input checked="" type="checkbox"/>	LED
<input checked="" type="checkbox"/>	<input type="checkbox"/>	LIGHT
<input checked="" type="checkbox"/>	<input type="checkbox"/>	LCD
<input type="checkbox"/>	<input checked="" type="checkbox"/>	CAN_T

**Figura 58:** Impostazioni dei Jumper per utilizzare il display LCD.

Si noti che si sono disattivati i LED e si è attivato il Display. Lasciando attivati i LED si vedrà il flusso dati sul display.

Vediamo ora come impostare la libreria nel caso in cui la si voglia utilizzare in sistemi differenti. All'interno del file LCD\_44780 sono dichiarate le seguenti costanti:

```
#define LCD_D0 LATDbits.LATD4
#define LCD_D1 LATDbits.LATD5
#define LCD_D2 LATDbits.LATD6
#define LCD_D3 LATDbits.LATD7
#define LCD_RS LATDbits.LATD2
#define LCD_E LATDbits.LATD3
#define LCD_RW LATDbits.LATD1
#define LCD_LED LATCbits.LATC1
```

In particolare ognuna di esse fa riferimento ad un pin del PIC. E' proprio questo riferimento che deve essere aggiornato; si noti che la modalità supportata è solo quella a 4 bit. LCD\_D0 ovvero il bit D0 è rappresentato in realtà dal bit D4 del Display, infatti i bit del display D0-D3 non sono utilizzati, quelli utilizzati in modalità 4 bit sono solo i bit D4-D7. Il bit RW non viene utilizzato dalla libreria se non per il fatto che viene posto a 0; per tale ragione in molte applicazioni tale pin è collegato a massa. Per non escludere applicazioni in cui si voglia però anche leggere dal display Freedom II supporta il controllo del bit RW. L'ultimo bit da impostare è il bit LCD\_LED ovvero il bit che controlla il LED di retroilluminazione.

Una Volta variata la libreria secondo le proprie esigenze è necessario ricompilare la libreria. In questo caso la compilazione genererà i messaggi di warning senza possibilità di eliminarli se non eliminando il controllo sulla definizione del nome LCD\_DEFAULT. Come ultimo passo sarà necessario impostare i pin utilizzati per il display LCD in maniera opportuna, ovvero come uscite<sup>118</sup>.

Vediamo ora le funzioni che sono offerte dalla libreria LCD\_44780.h.

Funzione	Descrizione
void <b>OpenLCD</b> (unsigned char)	Permette d'inizializzare il display LCD.
void <b>ClearLCD</b> (void)	Pulisce le righe del Display.
void <b>CursorLCD</b> (char,char)	1=ON cursor 0=OFF cursor; 1=ON blinking 0=OFF Blinking.
void <b>BacklightLCD</b> (char)	Accende e spegne il LED di retroilluminazione.
void <b>HomeLCD</b> (void)	Riposiziona il cursore all'inizio del display.
void <b>Line2LCD</b> (void)	Posiziona il cursore all'inizio della seconda riga.
void <b>ShiftLCD</b> (char,char)	Trasla le righe a destra o sinistra per un numero di volte assegnabile.
void <b>ShiftCursorLCD</b> (char,char)	Sposta il cursore a destra o sinistra per un numero di volte assegnabile.
void <b>WriteCharLCD</b> (unsigned char)	Scriva un carattere sul display.
void <b>WriteVarLCD</b> (unsigned char *)	Scriva una variabile sul display.
void <b>WriteIntLCD</b> (int, char)	Scriva un intero sul display, convertendo l'intero in stringa.
void <b>WriteStringLCD</b> (const rom char *)	Scriva una stringa costante sul display.

**Tabella 9:** Funzioni disponibili nella libreria LCD\_44780

Vediamo con maggior dettaglio le singole funzioni:

#### **void OpenLCD (unsigned char quartz\_frequency)**

Questa funzione deve essere eseguita una sola volta e sempre prima d'iniziare ad utilizzare le altre funzioni. Lo scopo della funzione è inizializzare il display, pulire le righe, posizionare il cursore all'inizio e togliere il suo lampeggio. Una mancata esecuzione di tale funzione lascia la prima riga del display scura. Si noti che il parametro da passare alla libreria rappresenta il valore della frequenza del quarzo del Clock utilizzato dal PIC; in questo modo è possibile gestire propriamente i vari impulsi che è richiesto avere per comandare il display. La funzione

<sup>118</sup> Si fa notare che se si utilizzasse la modalità di lettura del controllore, potrebbe essere necessario cambiare il valore di alcuni pin da output ad input. La libreria non supporta la modalità di lettura, dunque tutti i pin vanno impostati come output.

---

fa uso della libreria delay, dunque solo valori interi del quarzo sono supportati. Nel caso si abbiano valori frazionali è bene approssimare il valore all'intero più vicino per eccesso<sup>119</sup>.

**Parametri:**

**quartz\_frequency:** Valore della frequenza del quarzo o del Clock utilizzato.

**Ritorna:**

void.

**Esempio:**

```
// Inizializzo il Display LCD con PIC a 20MHz
OpenLCD (20);
```

---

**void ClearLCD (void)**

Tale funzione quando richiamata permette di ripulire il display da ogni scritta.

**Parametri:**

void.

**Ritorna:**

void.

**Esempio:**

```
// Ripulisco il Display LCD
ClearLCD ( );
```

---

**void CursorLCD (char active, char blinking)**

Questa funzione permette d'impostare alcune caratteristiche del cursore che punta la posizione in cui sarà scritto il prossimo carattere. Il primo valore passato tra parentesi attiva o disattiva il cursore; il valore 0 disattiva il cursore il valore 1 lo attiva. Il secondo valore attiva il lampeggio o meno del cursore, 0 lo disattiva 1 lo attiva.

**Parametri:**

**active:** Se vale 1 il cursore viene visualizzato, mentre e vale 0 il cursore è disattivo. Per attivare e disattivare il cursore si possono utilizzare le costanti TURN\_ON e TURN\_OFF.

**blinking:** Se vale 1 il cursore viene fatto lampeggiare, mentre e vale 0 il cursore non lampeggia. Per attivare e disattivare il lampeggio del cursore si possono utilizzare le costanti BLINK\_ON e BLINK\_OFF.

---

<sup>119</sup> Approssimando per difetto si rischierebbe di non far funzionare la libreria, poiché gli impulsi di comando potrebbero essere troppo veloci.

---

**Ritorna:**

void.

**Esempio:**

```
// Non visualizza il cursore e non effettua lampeggi  
CursorLCD (TURN_OFF, BLINK_OFF);
```

---

**void BacklightLCD (char active)**

Questa funzione permette di accendere o spegnere il LED di retroilluminazione. Quando vale 0 il LED è spento mentre quanto vale 1 il LED è acceso.

**Parametri:**

**active:** Se vale 1 il LED di retroilluminazione viene attivato mentre se vale 0 il LED di retroilluminazione viene spento. Per attivare e disattivare il LED di retroilluminazione si possono utilizzare le costanti TURN\_ON e TURN\_OFF.

**Ritorna:**

void.

**Esempio:**

```
// Accendo il LED di retroilluminazione  
BacklightLCD (TURN_ON);
```

---

**void HomeLCD (void)**

Tale funzione riposiziona il cursore alla prima riga in modo da iniziare a scrivere dall'inizio, sovrascrivendo i caratteri presenti.

**Parametri:**

void.

**Ritorna:**

void.

**Esempio:**

```
// Riporto il cursore alla posizione iniziale  
HomeLCD ();
```

---

**void Line2LCD (void)**

Tale funzione posiziona il cursore all'inizio della seconda riga.

---

**Parametri:**

void.

**Ritorna:**

void.

**Esempio:**

```
// Posizione il cursore all'inizio della seconda linea  
Line2LCD ();
```

---

**void ShiftLCD (char shift, char number\_of\_shift)**

Tale funzione trasla verso destra o verso sinistra, di un carattere, le righe del display, creando l'effetto di scorrimento. Dal momento che il display possiede una memoria interna ciclica una volta che i caratteri scompaiono dal display ritornando indietro verranno nuovamente visualizzati.

**Parametri:**

**shift:** Se vale 1 il testo sul display viene spostato a destra di una posizione, mentre se vale 0 viene spostato a sinistra di una posizione. Per spostare a destra o sinistra la scritta del display si possono usare le costanti LEFT, RIGHT.

**number\_of\_shift:** Specifica il numero di volte che deve essere effettuato lo shift nel verso scelto.

**Ritorna:**

void.

**Esempio:**

```
// Traslo il display di un carattere a sinistra  
ShiftLCD (LEFT,1);
```

---

**void ShiftCursorLCD (char shift, char number\_of\_shift)**

Per mezzo di questa funzione è possibile spostare la posizione attuale del cursore influenzando la posizione in cui verrà inserito il prossimo carattere o stringa.

**Parametri:**

**shift:** Se vale 1 il cursore, ovvero il nuovo punto di scrittura, viene spostato a destra di una posizione, mentre se vale 0 viene spostato a sinistra di una posizione. Per spostare a destra o sinistra il cursore si possono usare le costanti LEFT, RIGHT.

**number\_of\_shift:** Specifica il numero di volte che deve essere effettuato lo shift nel



---

verso scelto.

**Ritorna:**

void

**Esempio:**

```
// Sposto il cursore un carattere a destra  
ShiftCursorLCD (RIGHT,1);
```

---

**void WriteCharLCD (unsigned char value)**

Per mezzo di questa funzione è possibile scrivere un carattere ASCII sul display. Se viene passato un intero, verrà scritto il valore ASCII corrispondente all'intero.

**Parametri:**

**value:** Carattere da scrivere

**Ritorna:**

void.

**Esempio:**

```
// Scrivo il carattere M  
WriteCharLCD ('M');
```

---

**void WriteVarLCD (unsigned char \* buffer)**

Per mezzo di questa funzione, che al suo interno fa uso della funzione precedente, è possibile scrivere una stringa (Array di caratteri) sul display. La stringa di caratteri deve avere come ultimo elemento il valore speciale '\0'. La variabile in ingresso alla funzione è il puntatore all'inizio dell'Array ovvero il nome dell'Array.

**Parametri:**

**buffer:** Indirizzo dell'Array di caratteri contenente una stringa. L'Array deve terminare con il carattere speciale '\0'.

**Ritorna:**

void.

**Esempio:**

Si vedano gli esempi di fine Capitolo.

---

**void WriteStringLCD(const rom char \* buffer)**

---

Per mezzo di tale funzione è possibile scrivere sul display una stringa costante come potrebbero essere i messaggi per un menù da visualizzare. Si noti che scrivere una stringa costante è differente dallo scrivere una stringa contenuta all'interno di una variabile di tipo Array. Per tale ragione sono presenti due funzioni differenti.

**Parametri:**

**buffer:** Stringa contenuta in doppi apici o indirizzo di memoria rom.

**Ritorna:**

void

**Esempio:**

```
// Scrivo una stringa costante
WriteStringLCD ("Hello World");
```

Si osservi che in questo caso si è fatto uso del doppio apice e non dell'accento, invece richiesto nella scrittura del singolo carattere.

---

**void WriteIntLCD (int value, char number\_of\_digits)**

Questa funzione ritorna particolarmente utile quando si voglia scrivere un numero intero direttamente sul display. I numeri infatti non possono essere scritti direttamente sul display se non previa conversione in stringa. La funzione si occupa di tutto questo, dunque è possibile passare un semplice intero.

**Parametri:**

**value:** Valore intero da scriver sul display LCD

**number\_of\_digits:** Numero di cifre che si vuole visualizzare [0..5].

**0:** Il numero viene scritto con giustificazione a sinistra, utilizzando un numero di cifre variabile a seconda del numero stesso, ma fino ad un massimo di 5.

**1-5:** Il numero viene visualizzato con giustificazione a destra utilizzando un numero di digit pari a quello specificato, In questo modo il numero non cambia di lunghezza. Se il numero dovesse eccedere il numero di digit impostati, le cifre meno significative verranno perse. Il meno viene trattato come un digit.

**Ritorna:**

void.

**Esempio:**

```
int data = 128;
// Scrivo la variabile data sul display
WriteIntLCD (data,0);
```

---

## Leggiamo finalmente Hello World

Dopo questa carrellata d'informazioni vediamo come utilizzare il display. L'utilizzo della libreria per mezzo di esempi risulterà molto semplice, molto più che non descriverla nel suo insieme. Iniziamo con un primo esempio in cui risalutiamo il mondo...questa volta lo leggeremo!

```
#include <p18f4550.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF         Disabilitato il watchdog timer
//LVP = OFF         Disabilitato programmazione LVP
//PBADEN = OFF      Disabilitato gli ingressi analogici

void main (void){

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi, RC1 come output
    LATC = 0x00;
    TRISC = 0b11111101;

    // Imposto PORTD tutte uscite, RD0 come ingresso
    LATD = 0x00;
    TRISD = 0b00000001;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializzo il display LCD con quarzo a 20MHz
    OpenLCD (20);

    WriteStringLCD ("Hello World");

    BacklightLCD (TURN_ON);

    ShiftLCD (RIGHT,2);

    // Ciclo infinito
    while (1) {

    }
}
```

Si osservi che la libreria è stata inclusa definendo il nome LCD\_DEFAULT, in modo da

---

eliminare i messaggi di warning. In particolare questo è lecito poiché si sta utilizzando la scheda di sviluppo Freedom II.

```
#define LCD_DEFAULT
#include <LCD_44780.h>
```

Tra le impostazioni dei pin non c'è nulla di nuovo, ma si osservino le impostazioni dei pin della PORTD e PORTC.

```
// Imposto PORTC tutti ingressi, RC1 come output
LATC = 0x00;
TRISC = 0b11111101;

// Imposto PORTD tutte uscite, RD0 come ingresso
LATD = 0x00;
TRISD = 0b00000001;
```

Una volta inizializzati i pin è possibile inizializzare il nostro display LCD e scrivere il messaggio “Hello World”.

```
OpenLCD (20);

WriteStringLCD ("Hello World");

BacklightLCD (TURN_ON);

ShiftLCD (RIGHT,2);
```

Come visibile, grazie all'utilizzo della libreria bastano poche linee di codice. In particolare oltre a scrivere “Hello World” si è accesa la retroilluminazione. Se il messaggio non dovesse comparire si provveda a regolare il contrasto in maniera opportuna e si controlli che il display LCD sia propriamente abilitato.

L'ultima riga serve per spostare la scritta e centrarla nello schermo. In realtà questo comando non è fondamentale, si sarebbero potuto evitare. Per centrare la scritta si sarebbero potuti direttamente scrivere i due spazi ovvero “ Hello World” invece di “Hello World”; riducendo ulteriormente il codice.

Vediamo un secondo esempio in cui si fa uso di una struttura per memorizzare il nome e il cognome di una persona e si effettua una piccola manipolazione di Array. Questa “piccola” manipolazione fa di questo programma uno dei più complicati, visto che si introdurrà il concetto di puntatore che è tra gli aspetti più difficili per chi affronta il C per la prima volta.

```
#include <p18f4550.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF         Disabilitato il watchdog timer
//LVP = OFF         Disabilitato programmazione LVP
```

```

//PBADEN = OFF    Disabilito gli ingressi analogici

// Prototipo di funzione
void copy (unsigned char * dest, rom const char * parola);

typedef struct {
    unsigned char nome [20];
    unsigned char cognome [20];
} persona;

void main (void){

    // Variabile tizio è di tipo persona
    persona tizio;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi, RC1 come output
    LATC = 0x00;
    TRISC = 0b11111101;

    // Imposto PORTD tutte uscite, RD0 come ingresso
    LATD = 0x00;
    TRISD = 0b00000001;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializzo il display LCD con quarzo a 20MHz
    OpenLCD (20);

    BacklightLCD (TURN_ON);

    // Scrivo i dati nella variabile tizio
    copy (tizio.nome, "Mauro");
    copy (tizio.cognome, "Laurenti");

    // Scrivo il nome sull'LCD
    WriteVarLCD (tizio.nome);

    // Mi sposto alla seconda linea
    Line2LCD ();

    // Scrivo il cognome sull'LCD
    WriteVarLCD (tizio.cognome);

    // Ciclo infinito
    while (1) {

    }
}

```

```

void copy (unsigned char *dest, rom const char *parola) {
    while(*parola) {
        // Copio il carattere dentro l'array
        *dest = *parola;

        // Incremento il puntatore
        parola++;
        dest++;
    }
    // Inserisco il carattere di fine stringa
    *dest = '\0';
}

```

L'inizio del programma è simile al primo esempio, nulla di nuovo. In particolare oltre alle inclusioni dei file già visti si dichiara il prototipo di una funzione, che per ora non spiegheremo ed una struttura dati persona. Riguardo alla funzione basti sapere che viene utilizzata per copiare una stringa costante, tipo “Ciao Mamma” all'interno del nostro Array di caratteri.

```

// Prototipo di funzione
void copy (unsigned char * dest, rom const char * parola);

typedef struct {
    unsigned char nome [20];
    unsigned char cognome [20];
} persona;

```

Si noti che i campi nome e cognome sono degli Array di caratteri, ovvero delle stringhe. Diversamente dal C++, il C non supporta infatti la variabile di tipo `string`. All'inizio del `main` viene utilizzata proprio questa struttura per dichiarare una variabile `tizio` di tipo `persona`:

```

// Variabile tizio è di tipo persona
persona tizio;

```

Il programma diviene poi come il precedente, ovvero si inizializzano i vari pin del PIC in modo da poter utilizzare propriamente la libreria e si inizializza il display LCD. Successivamente viene richiamata la funzione `copy`, utilizzata per copiare una stringa all'interno della nostra struttura.

```

// Scrivo i dati nella variabile tizio
copy (tizio.nome, "Mauro");
copy (tizio.cognome, "Laurenti");

```

La ragione per cui deve essere richiamata la funzione è legata al fatto che il C, non supportando in maniera nativa le stringhe, non permette istruzioni del tipo:

```

nome = "Piero"

```

Per poter scrivere un nome o una qualunque frase all'interno di un Array è necessario

---

scrivere elemento per elemento, ovvero carattere, il nome o frase. Per agevolare il programmatore la Microchip fornisce la libreria standard C `string.h`, questa contiene varie funzioni ad hoc per le stringhe. In questo programma di esempio ho preferito scrivere la funzione `copy` piuttosto che usare la libreria `string.h` in modo da comprendere come poter manipolare un Array di caratteri.

Come detto la variabile `tizio` possiede i due campi `nome` e `cognome`, quindi sarà possibile accedere i singoli caratteri di questi due campi per mezzo di questa sintassi:

```
a = tizio.nome[2];  
b = tizio.cognome[3];
```

Questo permette di copiare nelle variabili `a` e `b` rispettivamente il terzo e il quarto carattere dei due campi `nome` e `cognome` (si ricordi che il primo carattere di un Array è alla posizione 0), `a` e `b` devono essere due variabili dichiarate come caratteri:

```
unsigned char a;  
unsigned char b;
```

Oltre a quanto scritto in precedenza è possibile anche scrivere:

```
d = tizio.nome;
```

Questa volta `d` non deve essere semplicemente un carattere! Infatti con questa sintassi senza parentesi quadre si intende l'indirizzo di memoria dove inizia il nostro Array<sup>120</sup> `tizio.nome`. Più propriamente si dice che `d` deve essere un puntatore di tipo `char`, ovvero servirà per puntare, ovvero memorizzare, l'indirizzo di una stringa di caratteri di tipo `char`. Per poter dichiarare un puntatore ad una variabile si fa uso del simbolo `*` prima del nome della variabile stessa; dunque un puntatore a unsigned char sarà<sup>121</sup>:

```
unsigned char * d;
```

Una volta che si ha il puntatore lo si può usare anche in sostituzione della sintassi in cui si accede l'elemento dell'Array con le parentesi quadre. Supponiamo di voler scrivere MAURO dentro l'Array `tizio.nome`. Quello che bisogna fare è scrivere nel primo elemento dell'Array la `'M'`, nel secondo `'A'`, nel terzo `'U'` nel quarto `'R'` nel quinto `'O'` e nel sesto il carattere `'\0'`, ovvero il carattere di fine stringa<sup>122</sup>, questo lo si può fare nel seguente modo:

```
tizio.nome [0] = 'M';  
tizio.nome [1] = 'A';  
tizio.nome [2] = 'U';  
tizio.nome [3] = 'R';  
tizio.nome [4] = 'O';  
tizio.nome [5] = '\0';
```

o facendo uso del puntatore `d` precedentemente dichiarato:

```
d = tizio.nome; //d punta all'elemento tizio.nome[0]
```

---

<sup>120</sup> I questo caso si ha una struttura, ma la cosa sarebbe stata equivalente con un semplice Array di caratteri chiamato `nome` piuttosto che un Array `nome` interno alla struttura `tizio`.

<sup>121</sup> I puntatori possono anche essere di altro tipo a seconda del tipo di Array o strutture dati con cui si ha a che fare.

<sup>122</sup> Il carattere di fine stringa è particolarmente importante poiché tutte le funzioni di manipolazione di stringhe fanno uso di tale carattere per sapere quando si è giunti a fine stringa.

```

*d = 'M';

d++;          //d punta all'elemento tizio.nome[1]
*d = 'A';

d++;          //d punta all'elemento tizio.nome[2]
*d = 'U';

d++;          //d punta all'elemento tizio.nome[3]
*d = 'R';

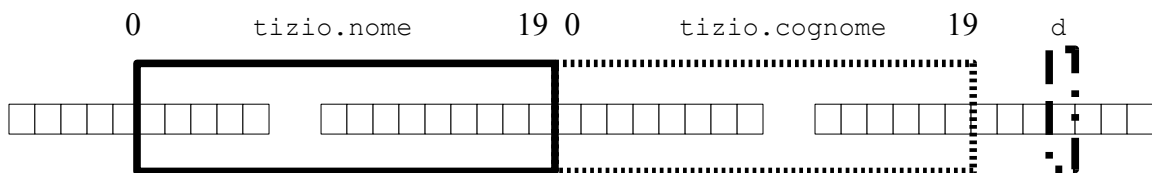
d++;          //d punta all'elemento tizio.nome[4]
*d = 'O';

d++;          //d punta all'elemento tizio.nome[5]
*d = '\0';

```

In questo esempio scrivere `*d` significa: scrivi nella variabile (elemento) puntata dall'indirizzo di memoria contenuto in `d`. Scrivere `d++` o comunque `d` uguale a qualcosa significa cambiare il valore del puntatore ovvero l'indirizzo puntato da `d`. Per mezzo di `d++` si incrementa l'indirizzo e dunque è come se si accedesse all'elemento successivo dell'Array.

Rivediamo il tutto con l'aiuto della Figura 59 in modo da comprendere l'argomento in maniera più chiara.



**Figura 59:** Esempio grafico di memoria RAM

Si consideri che ogni cella sia un Byte della memoria RAM dove sono contenute le nostre informazioni ovvero variabili. In particolare si consideri che le caselle dentro il rettangolo in grassetto continuo siano i 20 byte appartenenti all'Array `tizio.nome`, il rettangolo tratteggiato siano i 20 byte dell'Array `tizio.cognome` mentre il rettangolo punto linea sia la variabile puntatore a `unsigned char`. Ogni casella avrà un proprio indirizzo che il PIC utilizzerà per sapere dove andare a leggere e dove andare a scrivere un certo dato. Quando si scrive `d = tizio.nome` si scrive all'interno di `d` l'indirizzo della prima casella dell'Array `tizio.nome`. L'indirizzo però è solo un numero, per poter effettivamente andare a leggere o scrivere nella casella di memoria puntata dall'indirizzo contenuto in `d`, è necessario scrivere un asterisco prima di `d`. Senza mettere l'asterisco si accede al numero, contenuto in `d`, come se questa fosse una variabile normale. Dopo questa breve spiegazione ritorniamo al nostro programma, in particolare cerchiamo di capire come funziona la nostra funzione `copy`.

Si capisce che per far funzionare la nostra funzione è necessario indicare la posizione del nostro Array, dunque si passerà il suo indirizzo semplicemente scrivendo `tizio.nome` o `tizio.cognome`. Il passare l'indirizzo di una variabile è noto come passaggio di parametri per riferimento, si ricordi che i parametri passati ad una funzione avviene normalmente per valore. Come secondo campo sarà necessario passare la nostra stringa costante che contiene il nostro nome o il nostro cognome. Rivediamo la dichiarazione del prototipo della funzione `copy`:



---

```
void copy (unsigned char * dest, rom const char * parola);
```

E' possibile notare che la prima variabile della funzione `copy` rappresenta proprio un puntatore ad un Array, questo se si è seguito il ragionamento precedente spero non sorprenda. La seconda variabile che viene passata alla funzione è un po' più infelice, il tipo di variabile è un puntatore a caratteri costanti contenuti in rom, ovvero nella memoria programma (Flash). Si capisce che se la funzione avesse dovuto copiare un Array in un altro Array anche la seconda variabile sarebbe stata un puntatore a `char`; per questo caso bisogna scrivere dunque un'altra funzione. La funzione `copy` è la seguente:

```
void copy (unsigned char *dest, rom const char *parola) {  
  
    while(*parola) {  
  
        // Copio il carattere dentro l'array  
        *dest = *parola;  
  
        // Incremento il puntatore  
        parola++;  
        dest++;  
    }  
  
    // Inserisco il carattere di fine stringa  
    *dest = '\0';  
}
```

Al suo interno viene effettuato un ciclo `while` che termina quando il valore puntato dal puntatore `parola` vale `'\0'`, ovvero fine della stringa. Fino a che tale valore è diverso da tale carattere vengono eseguite le istruzioni interne al ciclo.

La prima istruzione nel ciclo copia il carattere puntato da `parola` nell'indirizzo puntato da `dest`, ovvero si copia un elemento dall'origine alla destinazione. L'istruzione successiva incrementa l'indirizzo contenuto nella variabile `parola` in modo da puntare il carattere successivo della parola d'origine. Allo stesso modo viene incrementato l'indirizzo contenuto nella variabile `dest` in modo da poter copiare il nuovo carattere nella cella successiva, il ciclo si ripete fino a che la parola non termina.

Usciti dal ciclo, all'ultimo elemento puntato da `dest`, si scrive il valore `'\0'`, infatti tale valore non viene trasferito poiché il ciclo `while` termina quando questo viene trovato all'interno della parola, come detto è fondamentale che questo valore si mantenga.

In ultimo, una volta che il nome e cognome sono caricati all'interno della nostra struttura, si possono scrivere semplicemente facendo uso della funzione di libreria `WriteVarLCD`, al quale viene passato il puntatore dell'Array:

```
// Scrivo il nome sull'LCD  
WriteVarLCD (tizio.nome);
```

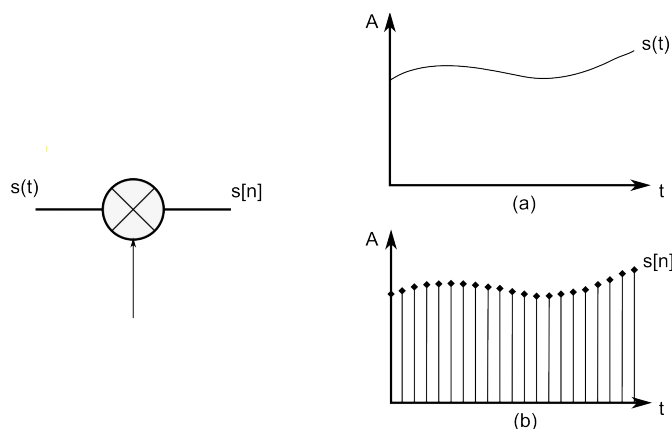
# Capitolo XI

## Utilizziamo il modulo ADC interno al PIC

Una delle applicazioni più importanti in cui vengono utilizzati i microcontrollori è quella di misurare un segnale proveniente da un sensore e prendere decisioni sulla base del valore rilevato. Tutto il mondo che ci circonda è analogico mentre il mondo del microcontrollore è digitale; per permettere al microcontrollore di comprendere una grandezza analogica è necessario dunque convertirla in digitale. In questo Capitolo dopo una breve introduzione sulle applicazioni dei convertitori analogici digitali (ADC) verrà introdotta la modalità di utilizzo del modulo ADC interno ai PIC. In ultimo verranno presentati diversi esempi di tipiche applicazioni in cui si utilizza l'ADC.

### Descrizione dell'hardware e sue applicazioni

Un convertitore ADC, ovvero *Analog to Digital Converter*, permette di tradurre una grandezza analogica in una grandezza digitale. Una grandezza analogica potrebbe essere la tensione di una batteria di cui si ha interesse sapere il valore. La caratteristica fondamentale di una grandezza analogica è quella di avere infiniti valori possibili, dunque una batteria di 12V che si scarica ad 11V passerà per 11.99999...999V poi 11.9999...998V fino ad arrivare ad 11V<sup>123</sup>. La capacità o meno di rilevare tante cifre dopo la virgola dipende solo dallo strumento che si sta utilizzando. Una grandezza digitale è caratterizzata da un numero finito di valori, per mezzo del quale viene descritta la grandezza d'interesse. Per esempio il nostro voltmetro digitale che possiede 3 digit (ovvero visualizza 3 cifre), per passare da 11 a 12 lo farà con un numero finito di valori. Da un punto di vista grafico, un segnale analogico è rappresentato per mezzo di una funzione reale continua. Un segnale digitale è invece una funzione discontinua, sequenza numerica, i cui valori sono rappresentati da un insieme finito e noto. In Figura 60 sono rappresentati due segnali:

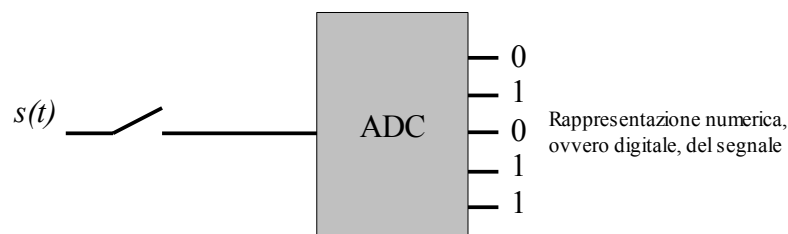


**Figura 60:** Esempio di segnale tempo continuo (a) e tempo discreto (b)

<sup>123</sup> Il numero di cifre non è in realtà infinito visto che la natura per sua natura è in realtà, a livello infinitesimale quantizzata...quasi digitale direi!

Il segnale analogico, è un segnale tempo continuo ed è normalmente contrassegnato dal nome  $s(t)$ , mentre la sua rappresentazione discontinua è contrassegnata da  $s[n]$ . Il segnale  $s[n]$  è come visibile una funzione discontinua ovvero una sequenza, ottenuta campionando, ovvero leggendo il valore del segnale d'interesse in tempi regolari  $nT$  dove  $T$  è il periodo di campionamento ed  $n$  è l' $n$ -esimo campione.

Il semplice campionamento genera in realtà una funzione discontinua ovvero tempo discreto, che non rappresenta ancora la rappresentazione digitale del segnale ovvero una sequenza numerica, solo quando i singoli campioni vengono tradotti in valori numerici, si ha un segnale digitale. L'assegnazione del valore numerico comporta una discretizzazione nell'ampiezza. La traduzione di un campione in valore numerico avviene per mezzo del convertitore analogico-digitale. Riassumendo, per convertire un segnale analogico tempo continuo in una grandezza digitale è necessario prima campionare il segnale d'interesse e convertire poi in valore numerico il campione. Da un punto di vista di schema a blocchi si ha:



**Figura 61:** Semplice rappresentazione di conversione analogico digitale

Nello schema a blocchi l'interruttore rappresenta quello che è noto come SH ovvero *Sample and Hold* (campiona e mantieni). La sua realizzazione viene generalmente ottenuta per mezzo di un interruttore realizzato in tecnologia MOS, per mezzo del quale si fa caricare un condensatore. L'interruttore viene aperto e chiuso ad intervalli regolari in modo da campionare il segnale e permettere successivamente la sua conversione. Spesso il SH è integrato nel modulo ADC, al quale dunque è possibile applicare direttamente il segnale. Quanto appena spiegato rappresenta una versione molto semplicistica di quello che sta dietro la teoria di conversione analogico digitale. In particolare non si è accennato alla teoria del campionamento visto che gli esempi che seguiranno sono una semplice lettura di tensioni piuttosto costanti nel tempo.

Il convertitore ADC che permette di convertire in valore numerico il nostro segnale può essere realizzato in molti modi a seconda delle applicazioni. Tra le tipologie più note si ricorda il convertitore flash, a rampa, a doppia rampa ed ad approssimazione successive<sup>1</sup>. In particolare i PIC18 possiedono un convertitore ad approssimazioni successive.

Ogni ADC è caratterizzato da alcuni parametri per mezzo dei quali può o meno essere idoneo per determinate applicazioni. Uno dei parametri principali è il numero di bit, ovvero il numero di valori per mezzo dei quali può rappresentare un segnale. Un ADC a 10bit potrà per esempio convertire una grandezza per mezzo di 10bit ovvero 1024 valori. La discretizzazione è normalmente di tipo lineare ovvero i vari valori sono tra loro equidistanti.

Un altro parametro importante dell'ADC è il valore di fondo scala, ovvero il valore massimo di tensione che può accettare. Dividendo il valore di fondo scala per il numero di valori per i quali l'ADC può convertire un segnale, si ha il valore minimo del segnale rilevabile, noto come quanto  $q$ . L'errore massimo che un ADC commette nel convertire un

---

segnale analogico in digitale è di  $q/2$ . Un ADC, a seconda della tipologia, converte un campione del segnale d'interesse in tempi differenti. In particolare l'ADC, al fine di poter essere sincronizzato con il mondo esterno possiede al suo interno della logica digitale per mezzo del quale effettua la conversione numerica e la pone sulle linee di uscita. Dal momento che un ADC è una macchina digitale (a stati) necessita di un Clock per poter svolgere la sua attività. Nel caso di un convertitore ad approssimazioni successive (presente nei PIC18) è necessario un tempo di conversione fisso e pari a  $N \cdot T_{CLOCK}$ , dove  $N$  è numero di bit del convertitore mentre  $T_{CLOCK}$  rappresenta il periodo del Clock utilizzato.

Oltre a quanto descritto un ADC è caratterizzato da molti altri parametri dinamici e non, che fanno di ogni modello la scelta migliore per determinate applicazioni. Per esempio la tensione operativa, la qualità della sorgente di riferimento, il numero di bit, ingresso single ended o differenziale, massima frequenza di campionamento, possono subito far decadere l'utilità di un ADC. Andando più in profondità un progettista controlla poi parametri quale ENOB (*Effective number of bit*), SFDR (*Spurious Free Dynamic Range*), SINAD (*Signal to Noise And Distortion*), SNR (*Signal to Noise Ratio*), FPBW (*Full Power Bandwidth*).

L'importanza dei vari parametri dinamici varia al variare delle applicazioni, in telecomunicazione l'SFDR è particolarmente utile come anche il SINAD. La descrizione di ogni parametro esula dagli scopi del testo, si rimanda alla bibliografia a fine testo per maggiori dettagli.

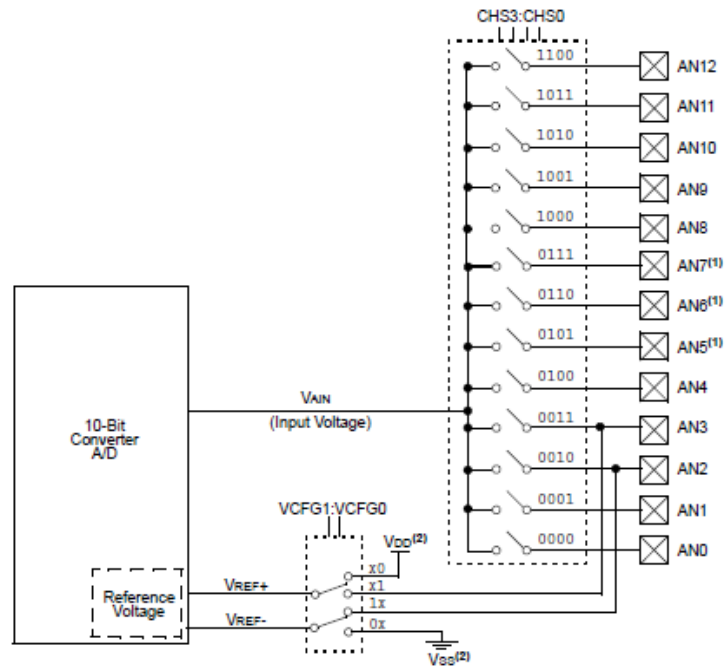
Da quanto appena visto, in ingresso al convertitore digitale si applicherà una tensione, qualora si voglia misurare una corrente sarà necessario prima convertirla in tensione. Un ADC lavora al meglio delle sue performance quando il nostro segnale occupa tutta la scala disponibile<sup>124</sup>, per tale ragione è sempre bene condizionare il segnale al fine di ottimizzare la sua ampiezza alla scala dell'ADC utilizzato. Frequentemente, scopo del condizionamento è anche quello di presentare una tensione all'ADC con un valore basso d'impedenza, ad esempio per il PIC18 è consigliato che la sorgente di tensione, ovvero il nostro segnale abbia un'impedenza non maggiore di 2.5KΩ. Per ridurre l'impedenza della sorgente si fa spesso uso di un amplificatore collegato come buffer, posto tra la sorgente e l'ADC. Nel caso della scheda di sviluppo Freedom II, per ragioni di semplicità non si è fatto uso di alcun buffer, ciononostante si possono comunque ottenere buoni risultati. Per maggiori informazioni sulla teoria del condizionamento del segnale si rimanda al Tutorial “Misure elettriche e tecniche di condizionamento del segnale” scaricabile dal sito [www.LaurTec.com](http://www.LaurTec.com).

Vediamo ora qualche dettaglio del modulo ADC presente all'interno del PIC18F4550. Lo schema a blocchi del modulo ADC è riportato in Figura 62. Si noti che il PIC18 pur avendo più ingressi analogici, a questi non corrisponde un modulo ADC. I PIC18 possiedono infatti un solo modulo ADC al cui ingresso viene presentato il segnale analogico di un solo ingresso analogico alla volta; sarà compito del Software selezionare l'ingresso di competenza.

Il numero di ingressi analogici varia da modello a modello, per esempio il PIC18F4550 possiede 13 ingressi, mentre la sua controparte a 28pin PIC18F2550 possiede solo 10 ingressi analogici (AN5, AN6, AN7 non sono presenti).

---

<sup>124</sup> In realtà si cerca di evitare di raggiungere full scale, spesso i datasheet specificano i vari parametri a -1dBFS ovvero un dB al disotto del fondo scala.



Note 1: Channels AN5 through AN7 are not available on 28-pin devices.  
2: I/O pins have diode protection to VDD and VSS.

**Figura 62:** Schema a blocchi del modulo ADC interno al PIC18F4550

## I registri interni per il controllo dell'ADC

Il modulo ADC è forse tra i moduli più funzionali e semplici dei PIC. Impostarlo ed utilizzarlo è questione di pochi passi per tale ragione nei programmi si fa spesso uso della scrittura diretta dei registri piuttosto che utilizzare delle librerie. L'utilizzo di una libreria può comunque risultare utile per quanto riguarda la leggibilità del codice. Per tale ragione dopo aver spiegato i vari registri si introdurrà in ogni modo la libreria standard offerta dalla Microchip.

I registri che è necessario impostare per il suo corretto utilizzo sono, escluso l'utilizzo eventuale delle interruzioni:

- ADDRESS
- ADDRESL
- ADCON0
- ADCON1
- ADCON2

I registri ADDRESS e ADDRESL sono utilizzati dal convertitore per caricare il valore numerico della conversione effettuata. Dal momento che il modulo ADC è a 10 bit mentre i due registri insieme formano un registro di 16 bit, si capisce che non tutti i bit sono utilizzati. In particolare è possibile giustificare il numero sia a destra che a sinistra a seconda delle esigenze. Gli altri registri sono invece di configurazione, il significato dei bit è piuttosto semplice, eccetto ADCON2 utilizzato per impostare il Clock dell'ADC. Per tale registro si spiegherà in maggior dettaglio il significato delle sue impostazioni. Vediamo qualche dettaglio dei vari registri facendo in particolare riferimento al PIC18F4550.

### Registro ADCON0: A/D Control Register 0

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
-	-	<b>CHS3</b>	<b>CHS2</b>	<b>CHS1</b>	<b>CHS0</b>	<b>GO/DONE</b>	<b>ADON</b>
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

#### Leggenda

R = Readable bit

-n = Value at POR

W = Writable bit

1 = Bit is set

U = Unimplemented bit read as 0

0 = Bit is cleared

S : Settable bit

x = Bit is unknown

**Bit 7-6** Non implementati. Sono letti come 0

**Bit 5-2** **CHS3:CHS0** : Selezione del canale Analogico

0000: Canale 0 (AN0)

0001: Canale 1 (AN1)

0010: Canale 2 (AN2)

0011: Canale 3 (AN3)

0100: Canale 4 (AN4)

0101: Canale 5 (AN5)

0110: Canale 6 (AN6)

0111: Canale 7 (AN7)

1000: Canale 8 (AN8)

1001: Canale 9 (AN9)

1010: Canale 10 (AN10)

1011: Canale 11 (AN11)

1100: Canale 12 (AN12)

1101: Non implementato

1110: Non implementato

1111: Non implementato

**Bit 1** **GO/DONE** : Stato della conversione

1: Avvia la conversione

0: La conversione è terminata

**Bit 0** **ADON** : Bit di controllo del modulo ADC

1: Il modulo ADC è attivato

0: Il modulo ADC è disattivato

Come detto ADCON0 risulta piuttosto di facile comprensione. Le configurazioni non implementate sono utilizzate in altri modelli di PIC18 in cui sono presenti 16 input analogici.

Si fa presente che il registro ADCON0 non deve essere scritto in una sola istruzione per attivare il modulo e avviare la conversione. Questo deve essere fatto in due istruzioni separate, ovvero prima si attiva il modulo ponendo ad 1 il bit 0, poi si avvia la conversione. I passi completi verranno comunque visti a breve.

## Registro ADCON1: A/D Control Register 1

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W <sup>1</sup>	R/W <sup>1</sup>	R/W <sup>1</sup>
-	-	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

### Leggenda

R = Readable bit  
-n = Value at POR

W = Writable bit  
1 = Bit is set

U = Unimplemented bit read as 0  
0 = Bit is cleared

S : Settable bit  
x = Bit is unknown

**Bit 7-6** Non implementati. Sono letti come 0.

**Bit 5** **VCFG1** : Configurazione VREF-  
1 : Ingresso AN2  
0 : Vss (massa)

**Bit 4** **VCFG1** : Configurazione VREF+  
1 : Ingresso AN3  
0 : Vcc

**Bit 3-0** **PCFG3:PCFG0** : Configurazione dei pin analogici e I/O

PCFG3:PCFG0	AN12	AN11	AN10	AN9	AN8	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
0000	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	D	D	A	A	A	A	A	A	A	A	A
0111	D	D	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Ingresso Analogico      D = I/O Digitale

1) Il valore POR (*Power On Reset*) dipende dal valore assunto dal bit di configurazione PBADEN.

PBADEN = 1 : PCFG,3:0> = 0000

PBADEN = 0 : PCFG,3:0> = 0111

Anche il registro ADCON1, come detto risulta piuttosto semplice. Si noti che per mezzo dei bit 4 e 5 è possibile impostare dei valori di riferimento esterni. Nelle nostre applicazioni si farà sempre uso di VSS e VDD, ovvero +5V e massa. L'utilità di utilizzare un riferimento esterno esula dagli scopi di questo testo. Si fa presente comunque che non è possibile impostare dei valori di riferimento superiori all'alimentazione utilizzata. Per maggior informazioni si faccia riferimento ai limiti operativi del PIC utilizzato.

I bit 0-3 servono per impostare i pin di cui si vuole far uso. Nonostante questo aspetto sia semplice, la sua impostazione può causare qualche problema. Di default, ovvero dopo il

Reset, tutti i pin con funzione ingresso analogico, sono impostati come analogici e non come digitali. Dunque, se si vogliono utilizzare i pin della PORTA come I/O digitali, sarà prima necessario impostare il registro ADCON1, anche se non si vuole usare il modulo ADC. Spesso questa impostazione viene ignorata ed i pin impostati come I/O digitali non funzionano “correttamente”.

Qualora si vogliano utilizzare i pin analogici sulla PORTB è necessario in fase di compilazione del programma impostare il registro di configurazione config:

```
//PBADEN = ON Disabilita gli ingressi analogici
#pragma config PBADEN = ON
```

Dal momento che i pin della PORTB sono assegnati come analogici in fase di compilazione, non potranno più essere utilizzati come digitali. Gli altri pin analogici possono invece essere utilizzati sia come analogici che digitali, cambiando la loro funzione durante l'esecuzione del programma.

Si noti che dalla Tabella di assegnazione pin, non è possibile assegnare un pin come analogico in qualunque combinazione. Se per esempio si volesse utilizzare il pin AN3, sarà comunque necessario avere impostati AN2 e AN1 come analogici. Questo comporta che durante la fase progettuale in cui si assegno i pin analogici è sempre bene fare le giuste considerazioni. Impostando i bit 0-3 tutti ad 1 si ha che tutti i pin sono assegnati come I/O digitali, ciononostante è sempre necessario impostare il registro config per il pin della PORTB.

#### Registro ADCON2: A/D Control Register

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	-	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Leggenda							
R = Readable bit		W = Writable bit		U = Unimplemented bit read as 0		S : Settable bit	
-n = Value at POR		1 = Bit is set		0 = Bit is cleared		x = Bit is unknown	

<b>Bit 7</b>	<b>ADFM</b> : Selezione formattazione di conversione 1 : Giustificazione a destra 0 : Giustificazione a sinistra
<b>Bit 6</b>	Non implementati. Sono letti come 0.
<b>Bit 5-3</b>	<b>ACQT2:ACQT0</b> : Tempo di acquisizione del modulo ADC 111 = 20 T <sub>AD</sub> 110 = 16 T <sub>AD</sub> 101 = 12 T <sub>AD</sub> 100 = 8 T <sub>AD</sub> 011 = 6 T <sub>AD</sub> 010 = 4 T <sub>AD</sub> 001 = 2 T <sub>AD</sub> 000 = 0 T <sub>AD</sub>
<b>Bit 2-0</b>	<b>ADCS2:ADCS0</b> : Clock del modulo ADC 111 = F <sub>RC</sub> (Clock RC Interno) 110 = F <sub>OSC</sub> /64 101 = F <sub>OSC</sub> /16 100 = F <sub>OSC</sub> /4 011 = F <sub>RC</sub> (Clock RC Interno) 010 = F <sub>OSC</sub> /32 001 = F <sub>OSC</sub> /8 000 = F <sub>OSC</sub> /2

Il registro ADCON2 richiede qualche spiegazione aggiuntiva, eccetto per il bit 7 che come

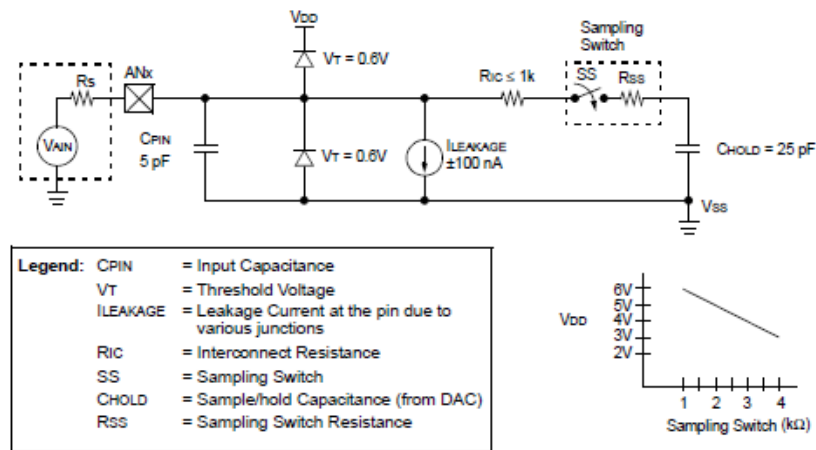


detto permette di formattare i registri ADDRESS e ADDRESL con giustificazione a destra o giustificazione a sinistra.

Per capire il significato dei bit 0-5 è necessario comprendere meglio il modello del pin analogico. E' infatti tale comprensione che permette di scegliere il valore ottimale dei bit 0-5.

Diciamo pure che mettendosi nel caso peggiore, ovvero ponendo i bit 3-5 tutti ad 1, e i bit 0-2 a 011 si sta più o meno coperti, senza però sapere il perché...

In Figura 63 è riportato il modello di ogni ingresso analogico, in particolare è anche visibile il generatore di tensione e la sua resistenza. Il generatore  $V_{AN}$  rappresenta il modello del nostro segnale  $s(t)$ .



**Figura 63:** Modello ingresso analogico

Alla destra del modello è presente il nostro interruttore che permette di campionare il segnale e far caricare il condensatore al valore assunto dal segnale in quel determinato momento; il modulo ADC provvederà poi al calcolo del valore numerico da associare. In questo modello il modulo ADC non è d'interesse.

Quando si applica una tensione ad un condensatore, spero sia noto che il condensatore si caricherà al valore di tensione a cui è stato sottoposto. Se tra il generatore di tensione ed il condensatore è presente una resistenza, il condensatore richiederà del tempo prima di caricarsi al valore finale, in particolare l'andamento della carica del condensatore è di tipo esponenziale. Si capisce che affinché il condensatore si carichi al valore d'interesse è necessario che l'interruttore rimanga chiuso un tempo minimo, che sarà funzione della resistenza totale presente nel percorso di carica. Non considerando la corrente di leakage, ovvero di perdita, la resistenza totale è pari alla somma delle resistenze  $R_S$ ,  $R_{IC}$ , ed  $R_{SS}$ . In particolare il tempo di acquisizione risulta pari a:

$$T_{ACQ} = \text{Amplifier Settling Time} + \text{Holding Capacitor Charging Time} + \text{Temperature Coefficient}$$

ovvero:

$$T_{ACQ} = T_{AMP} + T_C + T_{COFF}$$

Il valore  $T_{AMP}$  viene a dipendere dal settling time dell'operazionale eventualmente utilizzato come buffer<sup>125</sup>.

<sup>125</sup> Questa è almeno la mia interpretazione del datasheet. Non si capisce infatti dal modello se tale settling time appartenga ad un operazionale interno od esterno.

---

Il valore di  $T_C$  è pari a:

$$T_C = -C_{HOLD} \cdot (R_{IC} + R_{SS} + R_S) \cdot \ln \frac{1}{2048}$$

Il valore  $T_{COFF}$  espresso in  $\mu s$  è calcolato secondo la seguente formula:

$$T_{COFF} = (Temp - 25) \cdot 0.02$$

Per valori di temperatura  $Temp$  inferiori od uguali  $25^\circ C$   $T_{COFF}$  si considera pari a 0.

Considerando che:

$$C_{HOLD} = 25pF$$

$$R_S = 2.5K\Omega \text{ (trimmer montato su Freedom II)}$$

$$R_{SS} = 2K\Omega @ V_{CC} = 5V$$

$$R_{IC} = 1K\Omega \text{ (caso peggiore)}$$

$$Temp = 85^\circ C \text{ (temperatura per applicazioni industriali)}$$

Calcolando i vari termini si ha che:

$$T_{AMP} = 0 \mu s$$

$$T_C = 1.05 \mu s$$

$$T_{COFF} = 1.2 \mu s$$

dunque il tempo di acquisizione totale è pari a:

$$T_{ACQ} = T_{AMP} + T_C + T_{COFF} = 0 + 1.05 + 1.2 = 2.25 \mu s$$

Ora torniamo ai bit 0-2 del registro ADCON2. Questi bit stabiliscono la frequenza a cui il nostro ADC elaborerà le informazioni ovvero il suo Clock; tale periodo è definito come  $T_{AD}$ . Il PIC18F4550 come detto ha un ADC con approssimazioni successive dunque per effettuare la conversione richiederà un tempo pari  $10 T_{AD}$ . In realtà il tempo richiesto è di  $11 T_{AD}$  poiché un periodo viene utilizzato per scaricare il condensatore  $C_{HOLD}$ <sup>126</sup>. Come visibile dal registro ADCON2 il Clock può essere derivato dal Clock principale o da quello RC interno. Utilizzando il Clock RC interno è possibile far lavorare il convertitore anche quando il microcontrollore è in stato di SLEEP. Formalmente si potrebbe utilizzare la frequenza operativa più alta, in generale pari a  $F_{OSC}/2$ . Tale valore deve però essere scelto in maniera tale che il numero di  $T_{AD}$  impostati per il tempo di acquisizione, ovvero il valore dei bit 3-5, sia tale da poter garantire il valore  $T_{ACQ}$  precedentemente calcolato.

Considerando per esempio una frequenza operativa del PIC pari a 20MHz, selezionando  $F_{OSC}/2$  si ha che  $T_{AD}$  è pari a  $0.1 \mu s$ . Il massimo ritardo di acquisizione che è possibile avere è pari a  $20 T_{AD}$ , ovvero pari ad un ritardo di  $2 \mu s$ . Tale valore essendo inferiore al  $T_{ACQ}$  calcolato, vuol dire che  $F_{OSC}/2$  è troppo rapido. In particolare tra un'acquisizione e l'altra è necessario attendere  $3 T_{AD}$  che è bene includere nel tempo di acquisizione. Per entrare nelle specifiche si può dunque scegliere  $F_{OSC}/4$  ovvero  $T_{AD} = 0.2 \mu s$  e  $16 T_{AD}$  per il tempo di

---

<sup>126</sup> Questa è una tecnica che permette di ottimizzare il SH, facendo in modo che questo si trovi sempre a caricare il condensatore, piuttosto che far seguire a quest'ultimo il nuovo valore di tensione.

acquisizione.

Avendo calcolato il nostro tempo di acquisizione per una resistenza di sorgente pari a 2.5K $\Omega$ , si capisce che rallentando ulteriormente il tempo di acquisizione si potrebbero anche avere resistenze maggiori di 2.5K $\Omega$ . Si ricorda che il datasheet sconsiglia comunque di avere valori di  $R_s$  superiori a 2.5K $\Omega$ . Qualora il numero di non  $T_{AD}$  fossero sufficienti o si volesse comunque inserire un tempo di acquisizione personale, è possibile impostare i bit 3-5 del registro ADCON2 a 0. Questo valore comporta l'obbligo che prima di avviare una acquisizione, settando il bit GO/DONE, è necessario attendere il tempo di acquisizione richiesta dalla specifica applicazione.

Introdotti i vari registri vediamo di riassumere le impostazioni e passi da seguire nel caso in cui si voglia utilizzare il modulo ADC. I passi sono i seguenti:

1. Configurare i pin che si vuole come pin I/O o analogici (ADCON1)
2. Selezionare il canale che si vuole convertire (ADCON0)
3. Selezionare il tempo di acquisizione (ADCON2)
4. Selezionare il Clock di conversione (ADCON2)
5. Abilitare il modulo ADC (ADCON0)
6. Abilitare le interruzione se richieste (ADIF, ADIE, GIE)
7. Attendere il tempo di acquisizione (solo se si fa uso di ACQT2:ACQT0 = 000)
8. Avviare la conversione GO/DONE (ADCON0)
9. Attendere la fine della conversione (polling su GO/DONE o tramite interruzioni)
10. Leggere il valore della conversione (ADDRESH e ADDRESL)

Per effettuare nuove conversioni è possibile avviare nuove conversioni per mezzo del passo 8. Qualora si debba acquisire un altro canale è necessario selezionare il nuovo canale partendo al passo 2. Se tutti i canali hanno le stesse caratteristiche i passi possono essere riorganizzati in maniera da eseguire i passi 3-7 una sola volta.

Come visto i passi da seguire non sono molto complicati, ciononostante può ritornare utile l'utilizzo della libreria Microchip. Per utilizzare la libreria Microchip bisogna includere il file `adc.h`. Si fa presente che i vari PIC sono classificati in base alla versione del modulo ADC. Il PIC18F4550 appartiene al gruppo della versione 5. Le funzioni della libreria riportate in Tabella 10 potrebbero differire da quelle richieste qualora il PIC utilizzato appartenga ad un altro gruppo. Per vedere a che gruppo appartiene un particolare PIC si faccia riferimento alla documentazione della libreria che è possibile trovare nella directory `doc` del compilatore.

Funzioni	Descrizione
char <b>BusyADC</b> (void)	Controlla se la conversione è terminata o meno.
void <b>CloseADC</b> (void)	Disattiva il modulo ADC.
void <b>ConvertADC</b> (void)	Avvia la conversione del segnale.
void <b>OpenADC</b> (unsigned char <i>config</i> , unsigned char <i>config2</i> , unsigned char <i>portconfig</i> )	Attiva il modulo ADC con le relative impostazioni. Questa funzione varia a seconda delle versioni del modulo ADC. Questa è relativa alla versione 5 a cui appartiene il PIC18F4550.
int <b>ReadADC</b> (void)	Legge il risultato della conversione.
void <b>SetChanADC</b> (unsigned char channel)	Seleziona un canale di conversione.
void <b>SelChanConvADC</b> (unsigned char channel)	Seleziona un canale e avvia la conversione.

**Tabella 10:** Funzioni della libreria `adc.h`

---

### **char BusyADC (void)**

Per mezzo di questa funzione è possibile controllare lo stato di conversione del modulo ADC. In particolare la funzione ritorna il valore 1 se la conversione è ancora in corso, altrimenti ritorna il valore 0. Questa funzione può essere utilizzata per controllare la fine della conversione, quale alternativa all'utilizzo delle interruzioni. Al suo interno la funzione non fa altro che controllare il bit GO/DONE.

#### **Parametri:**

void.

#### **Ritorna:**

- 1: Il modulo ADC sta effettuando ancora la conversione
- 0: Il modulo ADC ha terminato la conversione

#### **Esempio:**

```
// Aspetta la fine della conversione
while (BusyADC ( ) );

// ...continua dopo la conversione
```

---

### **void CloseADC (void)**

Per mezzo di questa funzione è possibile disabilitare il modulo ADC precedentemente attivato. La funzione non fa altro che disattivare il bit ADON del registro ADCON0.

#### **Parametri:**

void.

#### **Ritorna:**

void.

#### **Esempio:**

```
// Chiude il modulo ADC
CloseADC ( ) ;
```

---

### **void ConvertADC (void)**

Per mezzo di questa funzione è possibile avviare la conversione del canale analogico precedentemente selezionato. Al suo interno la funzione non fa altro che settare il bit GO/DONE.

---

**Parametri:**

void.

**Ritorna:**

void.

**Esempio:**

```
// Avvia la conversione del canale analogico precedentemente selezionato  
ConvertADC();
```

---

**void OpenADC (unsigned char *config*, unsigned char *config2*, unsigned char *portconfig*)**

Per mezzo di questa funzione è possibile attivare il modulo ADC secondo le impostazioni volute. Il formato della funzione è differente a seconda della versione del modulo ADC considerato. Nel caso trattato si fa riferimento alla versione 5 presente nel PIC18F4550.

**Parametri:**

**config:** Tale parametro risulta un bitmask dei seguenti valori:

***Sorgente del Clock per il modulo ADC:***

ADC_FOSC_2	Fosc / 2
ADC_FOSC_4	Fosc / 4
ADC_FOSC_8	Fosc / 8
ADC_FOSC_16	Fosc / 16
ADC_FOSC_32	Fosc / 32
ADC_FOSC_64	Fosc / 64
ADC_FOSC_RC	Oscillatore RC interno

***Formato del dato:***

ADC_RIGHT_JUST	Giustificazione a destra
ADC_LEFT_JUST	Giustificazione a sinistra

***Selezione del tempo di acquisizione:***

ADC_0_TAD	0 Tad
ADC_2_TAD	2 Tad
ADC_4_TAD	4 Tad
ADC_6_TAD	6 Tad
ADC_8_TAD	8 Tad
ADC_12_TAD	12 Tad
ADC_16_TAD	16 Tad

**config2:** Tale parametro risulta un bitmask dei seguenti valori:

***Canale analogico:***

ADC_CH0	Canale 0
ADC_CH1	Canale 1
ADC_CH2	Canale 2
ADC_CH3	Canale 3
ADC_CH4	Canale 4
ADC_CH5	Canale 5
ADC_CH6	Canale 6
ADC_CH7	Canale 7
ADC_CH8	Canale 8
ADC_CH9	Canale 9
ADC_CH10	Canale 10
ADC_CH11	Canale 11
ADC_CH12	Canale 12
ADC_CH13	Canale 13
ADC_CH14	Canale 14
ADC_CH15	Canale 15

***Stato Interruzioni:***

ADC_INT_ON	Interruzioni abilitate
ADC_INT_OFF	Interruzioni disabilitate

**Selezione  $V_{REF+}$  e  $V_{REF-}$  :**

ADC_REF_VDD_VREFMINUS	$V_{REF+} = VDD$ e $V_{REF-} =$ Esterna
ADC_REF_VREFPLUS_VREFMINUS	$V_{REF+} =$ Esterna e $V_{REF-} =$ Esterna
ADC_REF_VREFPLUS_VSS	$V_{REF+} =$ Esterna e $V_{REF-} = VSS$
ADC_REF_VDD_VSS	$V_{REF+} = VDD$ e $V_{REF-} = VSS$

**portconfig:** Tale parametro è uno dei seguenti valori:

***Impostazione degli ingressi analogici e digitali:***

ADC_0ANA	Tutti digitali	
ADC_1ANA	Analogici:AN0	Digitali:AN1-AN15
ADC_2ANA	Analogici:AN0-AN1	Digitali:AN2-AN15
ADC_3ANA	Analogici:AN0-AN2	Digitali:AN3-AN15
ADC_4ANA	Analogici:AN0-AN3	Digitali:AN4-AN15
ADC_5ANA	Analogici:AN0-AN4	Digitali:AN5-AN15
ADC_6ANA	Analogici:AN0-AN5	Digitali:AN6-AN15
ADC_7ANA	Analogici:AN0-AN6	Digitali:AN7-AN15
ADC_8ANA	Analogici:AN0-AN7	Digitali:AN8-AN15
ADC_9ANA	Analogici:AN0-AN8	Digitali:AN9-AN15
ADC_10ANA	Analogici:AN0-AN9	Digitali:AN10-AN15
ADC_11ANA	Analogici:AN0-AN10	Digitali:AN11-AN15

---

ADC_12ANA	Analogici:AN0-AN11	Digitali:AN12-AN15
ADC_13ANA	Analogici:AN0-AN12	Digitali:AN13-AN15
ADC_14ANA	Analogici:AN0-AN13	Digitali:AN14-AN15
ADC_15ANA	Tutti analogici	

#### Ritorna:

void.

#### Esempio:

```
// Apertura del modulo ADC
OpenADC( ADC_FOSC_16 & ADC_RIGHT_JUST & ADC_8_TAD,
        ADC_CH0 & ADC_REF_VDD_VSS & ADC_INT_OFF,
        ADC_1ANA );
```

---

#### int ReadADC (void)

Per mezzo di questa funzione è possibile leggere il valore della conversione. Come detto tale valore è in realtà memorizzato all'interno di due registri di 8 bit. La funzione si preoccupa di unire i due registri e ritornare un unico valore sotto forma di intero. La funzione è così implementata:

```
int ReadADC(void) {
    return ( ( (unsigned int) ADRESH) <<8) | (ADRESL);
}
```

Si noti che viene effettuato un casting e poi traslato il registro ADRESH, al quale viene poi aggiunto il valore del registro ADRESL per mezzo dell'operatore bitwise *or* |. Tale funzione viene normalmente chiamata dopo che ci si è accertati che la conversione è terminata.

#### Parametri:

void.

#### Ritorna:

Ritorna il valore della conversione.

#### Esempio:

```
int valore_conversione = 0;

// Impostazioni ADC...
// ...

// Avvia la conversione del canale analogico selezionato
ConvertADC();

// Aspetta la fine della conversione
while (BusyADC());

// leggo il risultato della conversione
```

---

```
valore_conversione = ReadADC ();
```

---

### **void SetChanADC (unsigned char channel)**

Per mezzo di questa funzione è possibile impostare il canale analogico, ovvero pin, del quale si vuole effettuare la conversione. Tale funzione accetta vari parametri che possono essere al di fuori della validità del PIC utilizzato.

#### **Parametri:**

**channel:** Canale del quale si effettuerà la conversione.

ADC_CH0	Canale 0
ADC_CH1	Canale 1
ADC_CH2	Canale 2
ADC_CH3	Canale 3
ADC_CH4	Canale 4
ADC_CH5	Canale 5
ADC_CH6	Canale 6
ADC_CH7	Canale 7
ADC_CH8	Canale 8
ADC_CH9	Canale 9
ADC_CH10	Canale 10
ADC_CH11	Canale 11
ADC_CH12	Canale 12
ADC_CH13	Canale 13
ADC_CH14	Canale 14
ADC_CH15	Canale 15
ADC_CH_CTMU	Canale 13
ADC_CH_VDDCORE	Canale 14
ADC_CH_VBG	Canale 15

#### **Ritorna:**

void.

#### **Esempio:**

```
// Selezione AN2 per la conversione  
SetChanADC (ADC_CH2);
```

---

### **void SetChanConvADC (unsigned char channel)**

Questa funzione è praticamente identica alla precedente, ma oltre a selezionare il canale avvia anche la sequenza di conversione.

---



## Lettura di una tensione

In questo esempio si utilizzerà il modulo ADC, impostandolo un registro alla volta; dal momento in cui si ha a che fare con soli tre registri, il tutto risulterà piuttosto pratico. Per utilizzare propriamente Freedom II è necessario che i Jumper vengano impostati come riportato in Figura 64.

ON	OFF	JUMPER
		USB_CP
		USB_DET
		SCL
		SDA
		ANALOG
		INT
		TEMP
		SPK
		LED
		LIGHT
		LCD
		CAN_T

**Figura 64:** Impostazioni dei Jumper per la lettura del sensore Analog.

Ovvero si deve abilitare il trimmer nominato Analog e tenere i LED attivati mentre il modulo LCD risulta disattivo.

```
#include <p18f4550.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS      Impostato per lavorare ad alta frequenza
//WDT = OFF     Disabilito il watchdog timer
//LVP = OFF     Disabilito programmazione LVP
//PBADEN = OFF  Disabilito gli ingressi analogici

void main (void){

    // Variabile per salvare la sommatoria dei dati letti
    int sommatoria = 0;

    // Variabile per salvare il valore della conversione
    int lettura = 0;

    // Variabile utilizzata per il numero delle letture
    char i;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;
```

```

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

// Abilito A0-A1 come ingressi analogico
// VREF sono impostate a massa e VCC
ADCON1 = 0b00001101;

// Seleziono AN1 come ingresso
ADCON0 = 0b00000100;

// Imposto i tempi di conversione e giustificazione a destra
// TAD : FOSC/4
// TACQ: 16 TAD
ADCON2 = 0b10110100;

// Abilito l' ADC
ADCON0 |= 0b00000001;

// Ciclo infinito
while (1) {

    sommatoria = 0;

    // Effettuo 8 letture per fare una media
    for (i = 0; i < 8; i++) {

        // Avvio la conversione Analogico/Digitale
        ADCON0bits.GO = 1;

        // Attendo la fine della conversione
        while(ADCON0bits.GO);

        // Prelevo il valore della conversione
        lettura = (((int) ADRESH) << 8) | ADRESL;

        // Sommatoria delle letture fatte
        sommatoria = sommatoria + lettura;

    }

    // 3 shift per la media
    // 2 shift per la aver 8 bit
    sommatoria = sommatoria >> 5;

    // Visualizzo la lettura
    LATD = (char) (sommatoria);

}
}

```

Come è possibile vedere, facendo uso direttamente dei registri non è necessario includere nessuna libreria particolare. Dal momento che PORTB non è utilizzata per i suoi ingressi analogici, si è impostato:

```
#pragma config PBDEN = OFF
```

Le impostazioni del modulo ADC avvengono dopo aver impostato propriamente le porte, in particolare PORTA è stata definita con tutti ingressi. Questo non è differente dal solito, ma questa volta è importante che il pin AN1 sia impostato come input perché è quello al quale è collegato il trimmer del quale andremo a leggere la tensione<sup>127</sup>.

```
// Abilito AN0-AN1 come ingressi analogici
// VREF sono impostate a massa e VCC
ADCON1 = 0b00001101;

// Selezione AN1 come ingresso
ADCON0 = 0b00000100;

// Imposto i tempi di conversione e giustificazione a destra
// TAD : FOSC/4
// TACQ: 16 TAD
ADCON2 = 0b10110100;

// Abilito l' ADC
ADCON0bits.ADON = 0x01;
```

Spero che il codice non sia nulla di sorprendente. Si noti che per usare AN1 è necessario impostare anche AN0 come ingresso analogico. Una volta impostato il modulo si effettua la lettura dello stesso. Questo viene fatto in maniera continua all'interno del ciclo infinito `while`. Si noti in particolare che all'interno del ciclo `for` la lettura è ripetuta per 8 volte.

```
// Effettuo 8 letture per fare una media
for (i =0; i<8; i++){

    // Avvio la conversione Analogico/Digitale
    ADCON0bits.GO = 1;

    // Attendo la fine della conversione
    while(ADCON0bits.GO);

    // Prelevo il valore della conversione
    lettura = (((int) ADRESH) << 8) | ADRESL;

    // Sommatoria delle letture fatte
    sommatoria = sommatoria + lettura;
}
```

Il fatto di ripetere la lettura permette di calcolare un valore medio piuttosto che uno istantaneo che potrebbe creare delle fluttuazioni. Il numero di letture normalmente utilizzato per effettuare una media è normalmente una potenza di 2, in modo da poter poi fare la divisione per mezzo dello shift...che come visto potrebbe o meno creare ottimizzazioni!

La lettura del valore presente nei registri ADRESH e ADRESL, viene effettuata allo stesso modo della funzione `ReadADC ()`. Effettuate 8 letture si effettua la media dividendo per il numero delle letture. In particolare avendo effettuato 8 letture bisognerà effettuare 3 shift a

<sup>127</sup> Si rimanda alla documentazione di Freedom II per maggiori informazioni tecniche relative allo schema elettrico.

destra, ovvero  $2^3$ .

```
// 3 shift per la media
// 2 shift per la avere 8 bit
sommatoria = sommatioria >> 5;
```

Si noti che il numero di shift effettuati è però 5, ovvero due più del necessario. Questo è richiesto in maniera da poter avere un risultato ad 8 bit piuttosto che a 10 per poi visualizzare il risultato sulla stringa a LED.

## Lettura della temperatura

Vediamo ora un secondo progetto in cui si visualizzerà la temperatura ambiente. Diversamente dal primo esempio la temperatura verrà visualizzata sul display LCD in modo da avere una facile lettura, piuttosto che scriverla in binario!

Il sensore di temperatura montato sulla scheda Freedom II è il sensore LM35D utilizzabile per temperature da 0 a 100 °C. Lo stesso sensore è disponibile anche in versioni per misurare temperature negative, ma la configurazione di Freedom II supporta solo temperature superiori o uguali a 0 °C<sup>128</sup>. Il sensore LM35D possiede un fattore di scala lineare pari a 10mV/°C, questo significa che per ogni aumento della temperatura di un grado centigrado la sua uscita aumenta di 10mV e viceversa in caso di diminuzione della temperatura. La sua accuratezza  $\pm 1/4$  °C quindi, comunque amplifichiamo il segnale non potremo mai scendere al di sotto di tale accuratezza. In ogni modo il termometro che stiamo per realizzare è piuttosto semplice ed avrà una accuratezza di circa 1 °C. In particolare non è presente nessuna circuiteria per il condizionamento del segnale dunque non ci si aspetterà maggiore accuratezza.

Il modulo ADC interno al PIC18F4550 è come detto a 10 bit, impostando la dinamica a 5V, ovvero ponendo  $+V_{REF}$  e  $-V_{REF}$  rispettivamente a  $V_{CC}$  e massa, si ha che il quanto è pari a 5/1024 ovvero 5mV. Questo significa che ogni due bit si hanno proprio i 10mV che stanno ad indicare 1 °C...questo esempio ci porta ad un conto piuttosto fortunato. Da quanto esposto si capisce che dividendo per 2 il valore letto dal convertitore digitale, ovvero la tensione letta in uscita al sensore LM35D, si otterrà proprio il valore della temperatura. Per testare propriamente il software sotto riportato, la posizione dei Jumper deve essere come l'esempio precedente.

```
#include <p18f4550.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

void main (void) {
```

<sup>128</sup> Per maggiori informazioni sulla circuiteria relativa al sensore LM35D si faccia riferimento alla scheda tecnica di Freedom II.

```

// Variabile per salvare la sommatoria dei dati letti
unsigned int sommatoria = 0;

// Variabile per salvare il valore della conversione
int lettura = 0;

// Variabile utilizzata per il numero delle letture
char i;

// Imposto PORTA tutti ingressi
LATA = 0x00;
TRISA = 0xFF;

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi e RC1 come uscita
LATC = 0x00;
TRISC = 0b11111101;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

// Inizializzazione LCD
OpenLCD (20);
BacklightLCD (TURN_ON);

WriteStringLCD (" Temp : ");
WriteCharLCD (223);
WriteCharLCD ('C');

ShiftCursorLCD (LEFT,5);

// Abilito AN0-AN1-AN2 come ingressi analogici
// VREF sono impostate a massa e VCC
ADCON1 = 0b00001101;

// Seleziono AN2 come ingresso
ADCON0 = 0b00001000;

// Imposto i tempi di conversione e giustificazione a destra
// TAD : FOSC/4
// TACQ: 16 TAD
ADCON2 = 0b10110100;

// Abilito l' ADC
ADCON0bits.ADON = 0x01;

// Ciclo infinito
while (1) {

    sommatoria = 0;

    // Effettuo 64 letture per fare una media
    for (i =0; i<64; i++){

```

```

        // Avvio la conversione Analogico/Digitale
        ADCON0bits.GO = 1;

        // Attendo la fine della conversione
        while(ADCON0bits.GO);

        // Prelevo il valore della conversione
        lettura = (((int) ADRESH) << 8) | ADRESL;

        // Sommatoria delle letture fatte
        sommatoria = sommatoria + lettura;
    }

    // 6 shift per la media
    sommatoria = sommatoria >> 6;

    // 1 shift per avere la temperatura in gradi
    // divido cioè per 2
    sommatoria = sommatoria >> 1;

    WriteIntLCD (sommatoria,2);

    ShiftCursorLCD (LEFT,2);

    // Aspetto 2 secondi
    delay_ms (2000);
}
}

```

Il software non è molto diverso dal precedente, si è aggiunto solo il seguente codice per inizializzare il display:

```

// Inizializzazione LCD
OpenLCD (20);
BacklightLCD (TURN_ON);

WriteStringLCD ("  Temp :  ");
WriteCharLCD (223);
WriteCharLCD ('C');

ShiftCursorLCD (LEFT,5);

```

Scrivere il carattere 223 permette di scrivere °. Tale valore può essere trovato nella tabella dei caratteri del Datasheet del controllore HT44780. Si noti che dopo la scrittura dei vari caratteri si è fatto uno spostamento verso sinistra del cursore, in modo da localizzarlo nelle xx del seguente esempio:

Temp : xx °C

In questo modo ogni volta che si aggiorna la temperatura di devono solo aggiornare solo le xx e non tutto il Display. Questa tecnica può ritornare comoda quando si hanno molte scritte di cui solo una piccola parte deve essere aggiornata; altrettanto valida risulta nel caso in cui si utilizzino display più grandi. In questo esempio riscrivere sempre tutto quanto non sarebbe stato un grande problema.

L'impostazione del modulo ADC è identica al caso precedente, però questa volta si sono abilitati come ingressi analogici AN0-AN2 e si è selezionato AN2 come ingresso per la misura. La lettura e la media è anche svolta come del caso precedente ma il numero della

---

media è stato portato a 64. Quando si hanno valori per la media così alti è sempre bene accertarsi che il registro che conterrà la sommatoria sia sempre delle dimensioni opportune per contenere la somma dei campioni richiesti. Oltre alla divisione per 64 ottenuta per mezzo di uno shift verso destra di 6 posizioni, è presente anche un secondo shift per ottenere il valore in gradi centigradi, questo è stato separato solo per chiarezza:

```
// 1 shift per avere la temperatura in gradi
// divido cioè per 2
sommatoria = sommatoria >> 1;
```

Lo scopo di questo shift è solo di dividere per due ed ottenere il valore della temperatura in gradi. Il valore della temperatura in gradi viene scritto per mezzo della funzione della libreria del display:

```
WriteIntLCD (sommatoria,2);
```

Questa permette di convertire in stringa un valore numerico intero ed assegna due digit con giustificazione a destra, al numero finale. Questo significa che potenzialmente si potrebbero misurare temperature fino a 99 °C. Una volta aggiornato il display si effettua uno shift a sinistra in maniera da riposizionare il cursore sulle xx della temperatura; si è inserito anche un ritardo di due secondi in maniera da rallentare le letture. Questo ritardo senza far nulla è accettabile in un'applicazione come questa in cui non si fa altro che leggere la temperatura, in sistemi più complessi tale ritardo potrebbe essere inaccettabile e l'utilizzo dei Timer sarebbe d'obbligo.

```
ShiftCursorLCD (LEFT,2);
// Aspetto 2 secondi
delay_ms (2000);
```

Il Display con la temperatura risulterà come in Figura 65.



**Figura 65:** Visualizzazione della temperatura

**Nota:**

Si raccomanda di fare solo misure di temperature ambientali 0-45 gradi. Temperature superiori richiederebbero considerazioni tecniche non trattate.

---

## Lettura dell'intensità luminosa

In quest'ultimo esempio si utilizza il modulo ADC per acquisire il segnale, ovvero la tensione, in uscita dal partitore di tensione composto dalla fotoresistenza PH e il resistore da 100K, presenti sul canale AN0 della scheda Freedom II. Il software è praticamente identico al precedente se non per aver selezionato il canale AN0.

```
#include <p18f4550.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF         Disabilitato il watchdog timer
//LVP = OFF         Disabilitato programmazione LVP
//PBADEN = OFF      Disabilitato gli ingressi analogici

void main (void){

    // Variabile per salvare la sommatoria dei dati letti
    unsigned int sommatoria = 0;

    // Variabile per salvare il valore della conversione
    int lettura = 0;

    // Variabile utilizzata per il numero delle letture
    char i;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0b11111101;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializzazione LCD
    OpenLCD (20);
    BacklightLCD (TURN_ON);

    WriteStringLCD ("    Lux : ");
```



```

// Abilito AN0-AN1-AN2 come ingressi analogici
// VREF sono impostate a massa e VCC
ADCON1 = 0b00001101;

// Seleziono AN0 come ingresso
ADCON0 = 0b00000000;

// Imposto i tempi di conversione e giustificazione a destra
// TAD : FOSC/64
// TACQ: 20 TAD
ADCON2 = 0b10111110;

// Abilito l' ADC
ADCON0bits.ADON = 0x01;

// Ciclo infinito
while (1) {

    sommatoria = 0;

    // Effettuo 32 letture per fare una media
    for (i =0; i<32; i++){

        // Avvio la conversione Analogico/Digitale
        ADCON0bits.GO = 1;

        // Attendo la fine della conversione
        while(ADCON0bits.GO);

        // Prelevo il valore della conversione
        lettura = (((int) ADRESH) << 8) | ADRESL;

        // Sommatoria delle letture fatte
        sommatoria = sommatoria + lettura;

    }

    // 5 shift per la media
    sommatoria = sommatoria >> 5;

    WriteIntLCD (sommatoria,4);

    ShiftCursorLCD (LEFT,4);

}
}

```

Tra le altre piccole variazioni apportate si osservi che il tempo di acquisizione ed il Clock sono stati posti al massimo, in modo da rallentare l'acquisizione. Questo è legato al fatto che l'impedenza della sorgente è rappresentata dal parallelo del resistore 100K e la fotoresistenza e può essere piuttosto alta. Per una misura più accurata si sarebbe potuto scegliere il controllo manuale del tempo di acquisizione, ma visto che la misura della luce effettuata in questo modo è piuttosto grossolana, parlare di accuratezza non ha comunque molto senso. In ogni modo in applicazioni di precisione si sarebbe dovuto utilizzare un buffer in modo da rendere la sorgente a bassa impedenza e disaccoppiare il modulo ADC dal sensore. Si fa presente che il livello di luminosità viene visualizzato come semplice numero senza unità di misura.

# Capitolo XII

## Utilizziamo l'EUSART interna al PIC

Comunicare è sempre alla base di ogni sistema embedded. Normalmente i sistemi embedded utilizzano la porta seriale RS232 o LAN per poter effettuare il Debug del sistema, durante la fase di sviluppo, o comunicare con quest'ultimo durante la normale esecuzione di un'applicazione. I PIC per la fase di Debug normalmente non utilizzano né la porta seriale né la LAN, ciononostante tali porte di comunicazione ricoprono un ruolo molto importante. In questo Capitolo si introdurrà la porta seriale e si svilupperanno alcune applicazioni d'esempio.

### Descrizione dell'hardware e sue applicazioni

Un qualunque dispositivo hardware dopo aver elaborato un segnale in maniera analogica o digitale deve poter comunicare per mezzo di un'interfaccia di output i risultati ottenuti. La visualizzazione dei dati o ulteriore elaborazione possono avvenire per mezzo di un altro dispositivo che può trovarsi fisicamente distante dal primo, sorge dunque il problema della trasmissione dei dati.

Per brevi tratte si tende a far prevalere una comunicazione parallela laddove effettivamente i dati da trasmettere siano digitali. Con trasmissione parallela si intende l'utilizzo di più linee dati<sup>129</sup> (fili) per trasmettere l'informazione. Si pensi ad esempio ai bus collegati al microprocessore, che permettono lo scambio d'informazione tra quest'ultimo e le varie periferiche sulla scheda madre; questo tipo di collegamento permette velocità particolarmente elevate. Nonostante le apparenti brevi distanze che separano il microprocessore dalle varie periferiche, è necessario attendere un tempo non nullo affinché tutti i bit che costituiscono l'informazione da trasmettere arrivino alla fine del bus. Per tale ed altre ragioni la trasmissione parallela non viene utilizzata per lunghe distanze, anche se in realtà la massima distanza viene a dipendere pure dalla massima frequenza con la quale si vuole trasmettere l'informazione. A tal proposito si pensi alla porta parallela utilizzata per la stampante, che può raggiungere distanze dei 3m senza molti problemi. Questo non sarebbe possibile se si volesse trasmettere le informazioni alla stampante alla stessa velocità con cui il microprocessore può leggere o scrivere nella memoria RAM.

Per lunghe distanze si tende ad utilizzare una trasmissione seriale; per trasmissione seriale si intende che l'informazione può viaggiare su un'unica linea dati<sup>130</sup>. In questa classe di trasmissioni rientra lo standard RS232 che viene gestito per mezzo del modulo interno al PIC nominato EUSART (*Enhanced Universal Synchronous Asynchronous Receiver Transmitter*). Normalmente i nomi dei moduli per gestire la RS232 sono nominati USART o UART a seconda delle opzioni. Molti PIC gestiscono però alcune opzioni aggiuntive per migliorare il loro utilizzo in applicazioni particolari, per cui è stata aggiunta la E.

<sup>129</sup> Si è specificato linee dati poiché generalmente nella trasmissione dell'informazione sono presenti anche linee di controllo necessarie per la corretta trasmissione delle informazioni.

<sup>130</sup> Utilizzando segnali LVDS (*Low Voltage Differential Signal*) o differenziali di altro tipo, è possibile effettuare trasmissioni parallele a distanze piuttosto lunghe ma le problematiche tecniche associate al ritardo dei bit (*skew*) non sono poche, in ogni modo si cerca sempre di ridurre il numero di cavi necessari.

---

Sia che la trasmissione sia parallela o seriale si parla di trasmissioni simplex, half-duplex e full-duplex, con questi nomi si caratterizza ulteriormente il tipo di trasmissione. In particolare si ha una trasmissione simplex quando i dati viaggiano in un unico senso, ovvero si invia un'informazione senza preoccuparsi di nessuna risposta da parte del ricevitore. Con questo tipo di trasmissione non è dunque possibile sapere se l'informazione è effettivamente giunta a destinazione ed inoltre non c'è un vero scambio di informazioni<sup>131</sup>.

Per trasmissione half-duplex si intende una trasmissione che può avvenire in ambo i sensi ma non contemporaneamente. Il fatto che la trasmissione non debba avvenire contemporaneamente può essere dovuta alla presenza di un solo cavo per la trasmissione dati o poiché le risorse hardware non permettono di ottenere una comunicazione bidirezionale nello stesso intervallo di tempo.

In ultimo vi è la trasmissione full-duplex che permette una comunicazione in ambo i sensi e in contemporanea. Questo non necessariamente significa che siano presenti due linee dati, una per la trasmissione e una per la ricezione, infatti con la cosiddetta moltiplicazione di frequenza, utilizzata per esempio in telefonia, è possibile trasmettere più informazioni in contemporanea su un unico cavo.

Due altri sottogruppi di trasmissioni che è possibile individuare sia tra quelle parallele che quelle seriali è l'insieme delle trasmissioni sincrone e quelle asincrone. Per trasmissione sincrona si intende che assieme alle linee dati viene trasmesso anche un segnale di Clock in modo da sincronizzare le varie fasi di trasmissione e ricezione dati<sup>132</sup>, questo può o meno essere fornito per mezzo di linea dedicata. Con trasmissioni sincrone, avendo a disposizione una base dei tempi, è possibile raggiungere velocità di trasmissione più alte che non nel caso asincrono. Anche se letteralmente asincrono vuol dire senza sincronismo, non significa che questo non sia presente. Per asincrono si intende solo che il sincronismo non viene trasmesso per mezzo di un Clock presente su una linea ausiliaria, il sincronismo è sempre necessario.

Nelle trasmissioni asincrone, sia il trasmettitore che il ricevitore sono preimpostati a lavorare ad una certa velocità. Quando il ricevitore si accorge che il trasmettitore ha iniziato a trasmettere, conoscendo la sua velocità saprà interpretarlo<sup>133</sup>.

Per l'invio di dati sono presenti molti tipi di trasmissioni sia seriali che parallele, come anche in ogni lingua affinché i due interlocutori si possano capire è necessario che parlino la stessa lingua. Ogni idioma ha una struttura grammaticale che permette una corretta trasmissione dell'informazione. In termini tecnici si parla di protocollo per intendere l'insieme di regole e specifiche che il trasmettitore e il ricevitore devono avere affinché si possano comprendere. Le varie regole si traducono per lo più in impostazioni del PIC stesso.

Si fa presente che EUSART e protocollo RS232 sono due cose distinte, in particolare il protocollo RS232 impone alcune regole di tipo meccanico (connettore) ed elettrico al fine da garantire uno standard sul cosiddetto physical layer (livello fisico) ed anche sul formato dei dati. Il modulo EUSART rispetta il formato dei dati che il protocollo RS232 richiede, ma per poter rispettare l'intero protocollo è necessario aggiungere il connettore opportuno<sup>134</sup> e soprattutto un traslatore di livello (transceiver) per convertire i segnali TTL uscenti dal PIC in RS232 e viceversa. Lo stesso modulo EUSART potrebbe essere utilizzato anche per una trasmissione RS485, cambiando semplicemente il transceiver.

La scheda di sviluppo Freedom II è conforme allo standard RS232 dunque può essere

---

<sup>131</sup> Un esempio di trasmissione simplex che sfrutta i canali radio è rappresentata dalla televisione e dalla radio stessa. Questo tende ad essere sempre meno vero con l'avvento dei decoder e delle TV via cavo.

<sup>132</sup> Si veda per esempio il Tutorial sul Bus I<sup>2</sup>C.

<sup>133</sup> Per mezzo di alcuni accorgimenti è comunque possibile impostare il ricevitore alla frequenza del trasmettitore, senza conoscere a priori la frequenza di trasmissione.

<sup>134</sup> Nulla vieta di utilizzare altri connettori, ma per poter comunicare con periferiche che rispettano il protocollo, sarà necessario avere un adattatore.

utilizzata per una comunicazione diretta con il PC<sup>135</sup>. Per lo standard RS485 e il suo collegamento al modulo EUSART si faccia riferimento alla scheda Freedom (non Freedom II).

In Figura 66 è riportato un semplice schema che fa uso di un PIC a 28 pin e il transceiver RS232.

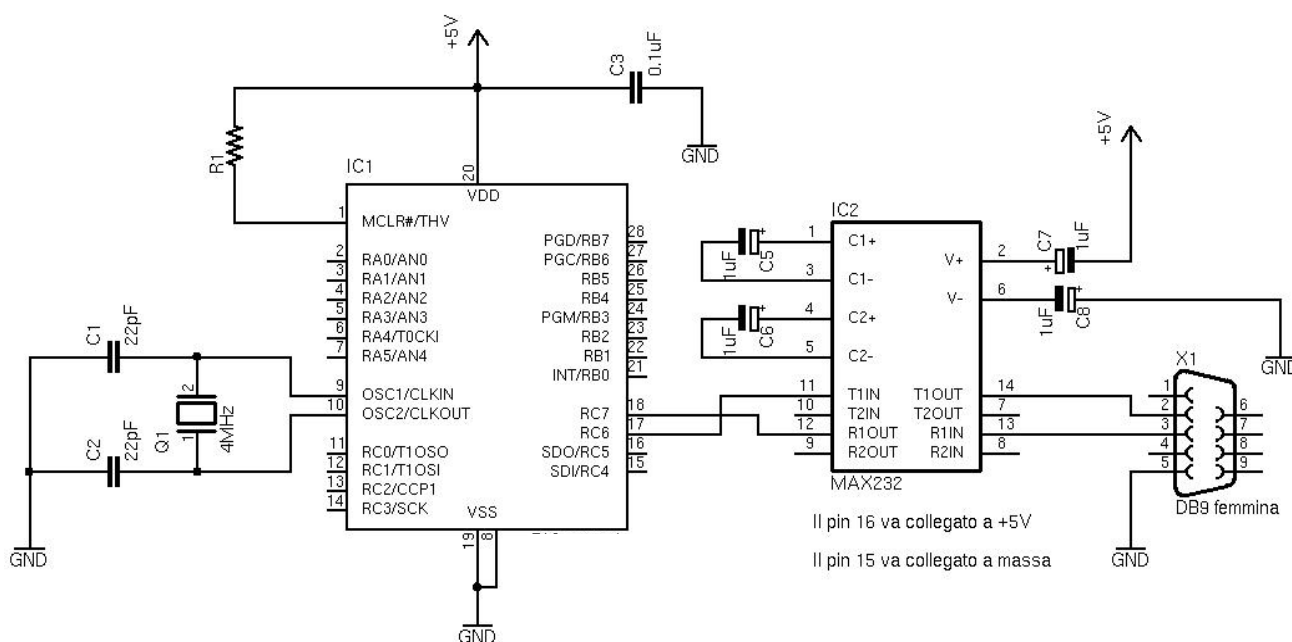


Figura 66: Schema d'esempio con un PIC e un transceiver RS232

Per ulteriori informazioni sul protocollo RS232 si faccia riferimento al Tutorial “Il protocollo RS232”, scaricabile dal sito [www.LaurTec.com](http://www.LaurTec.com).

## Impostare l'EUSART per un corretto utilizzo

Come detto il protocollo RS232 possiede molte regole per quanto riguarda il formato dei dati, il modulo interno EUSART le rispetta ed ha inoltre anche caratteristiche non previste dal protocollo RS232. Un esempio è l'utilizzo dell'indirizzamento, particolarmente utile qualora si faccia utilizzo di transceiver RS485. Questo risulta molto utile con questi tipi di transceiver poiché il protocollo RS485 permette di collegare su uno stesso bus più periferiche, dunque l'approccio di utilizzare indirizzi diversi per ogni periferica risulta molto importante. Nel caso del protocollo RS232, essendo pensato per una comunicazione *peer to peer* (punto punto), il concetto d'indirizzo non è molto importante dunque non verrà discusso ulteriormente. Piuttosto che spiegare i vari registri interni da utilizzare per impostare correttamente il modulo EUSART del PIC18F4550 a cui si farà riferimento, si spiegherà direttamente la libreria standard offerta dalla Microchip. Il file che è necessario includere è il file `usart.h`.

<sup>135</sup> Per ragioni di spazio i portatili non hanno più la porta seriale RS232, normalmente è richiesto un convertitore USB-RS232, ciononostante lo standard RS232 è ancora molto usato grazie alla sua facilità d'uso. Non a caso praticamente ogni PIC di ogni famiglia ha un modulo per supportare il protocollo RS232 (formato dati). Qualora il modulo non fosse presente potrebbe comunque essere facilmente implementato via software.

---

Vediamo un riassunto delle funzioni principali della libreria usart.h:

Funzioni	Descrizione
char <b>BusyxUSART</b> (void)	Controlla se l'USART è occupata.
void <b>ClosexUSART</b> (void )	Chiude l'USART.
char <b>DataRdyxUSART</b> (void)	Controlla se sono stati ricevuti dati.
void <b>OpenxUSART</b> (unsigned char ,unsigned int )	Inizializza l'USART.
char <b>ReadxUSART</b> ( void )	Legge un dato dal buffer di ricezione.
void <b>WritexUSART</b> (char )	Trasmette un dato in uscita.

**Tabella 11:** Funzioni principali della libreria standard usart.h

Si noti subito che ogni funzione possiede una x prima della parola USART finale. La x deve essere cambiata con il numero del modulo interno al PIC. I PIC18 possiedono infatti, a seconda del modello, fino a 3 USART interne. Nel caso di modelli con una sola USART, la x va tolta.

---

#### **char BusyUSART ( void )**

Per mezzo di questa funzione è possibile controllare lo stato di trasmissione dell'USART. La funzione ritorna il valore 1 se l'USART sta trasmettendo il dato altrimenti ritorna il valore 0. Questa funzione può essere utilizzata per controllare la fine della trasmissione di un byte.

#### **Parametri:**

void.

#### **Ritorna:**

- 1: Il modulo sta ancora trasmettendo il dato
- 0: Il modulo è libero per trasmettere altri dati

#### **Esempio:**

```
// Aspetta la fine della trasmissione dati
while (BusyUSART() );

// Continua dopo la trasmissione
```

---

#### **void CloseUSART ( void )**

Per mezzo di questa funzione viene chiuso il modulo l'USART precedentemente aperto.

#### **Parametri:**

void.

---

**Ritorna:**

void.

**Esempio:**

```
// Chiudi il modulo USART  
CloseUSART();
```

---

**char DataRdyUSART ( void )**

Per mezzo di questa funzione è possibile controllare se nel buffer di ricezione dell'USART è presente almeno un byte. Se è presente un dato viene ritornato il valore 1 altrimenti se non è presente nessun dato viene ritornato il valore 0.

**Parametri:**

void.

**Ritorna:**

1: Il modulo ha un byte nel buffer di ricezione

0: Non sono stati ricevuti dati

**Esempio:**

```
// Aspetta per la ricezione di dati  
while (!DataRdyUSART());  
  
// Continua dopo la ricezione del dato
```

---

**void OpenUSART ( unsigned char *config*, unsigned int *spbrg* )**

Per mezzo di questa funzione è possibile aprire il modulo USART ed impostare i parametri di trasmissione, ricezione ed eventuali interruzioni. Per fare questo bisogna riempire i due campi della funzione OpenUSART. Il primo valore è dato da un AND bitwise di varie costanti. Dal valore finale la funzione rileva le impostazioni della porta interna. Il secondo valore è un registro che permette d'impostare la frequenza di trasmissione.

**Parametri:**

**config:** Il primo valore della funzione viene impostato per mezzo delle seguenti costanti, unite tra loro per mezzo dell'AND bitwise &. Le costanti d'interesse sono:

**Interruzione di Trasmissione:**

USART_TX_INT_ON	Interruzione TX ON
USART_TX_INT_OFF	Interruzione TX OFF

**Interruzione in ricezione:**

---

USART_RX_INT_ON	Interruzione RX ON
USART_RX_INT_OFF	Interruzione RX OFF

#### Modalità USART:

USART_ASYNC_MODE	Modalità Asincrona
USART_SYNC_MODE	Modalità Sincrona

#### Larghezza dati:

USART_EIGHT_BIT	8-bit
USART_NINE_BIT	9-bit

#### Modalità Slave/Master:

USART_SYNC_SLAVE	Modalità Slave sincrona (si applica solo in modalità sincrona)
USART_SYNC_MASTER	Modalità Master sincrona (si applica solo in modalità sincrona)

#### Modalità di ricezione:

USART_SINGLE_RX	Ricezione singola
USART_CONT_RX	Ricezione multipla

#### Baud rate:

USART_BRGH_HIGH	Baud rate alto
USART_BRGH_LOW	Baud rate basso

**spbrg:** Il secondo valore da passare alla funzione è *spbrg* che permette di impostare la frequenza di trasmissione. Tale valore varia a seconda della frequenza del quarzo che si sta utilizzando e se si sta utilizzando un alto baud rate o meno. Alto baud rate si ha quando il flag BRGH è impostato ad 1 mentre un basso baud rate si ha con BRGH impostato a 0. Questi valori sono assegnati dalla funzione OpenUSART per mezzo delle costanti USART\_BRGH\_HIGH e USART\_BRGH\_LOW. Per decidere il valore della variabile *spbrg* si può far uso delle tabelle riportate sui datasheet del microcontrollore che si sta utilizzando. In Tabella 12 sono riportate quelle di maggior interesse ovvero per il caso asincrono alto baud rate e basso baud rate.

Le prime due Tabelle fanno riferimento all'opzione basso baud rate; ogni colonna delle tabelle fa riferimento a diverse frequenze di quarzo. Le ultime due Tabelle fanno riferimento al caso sia selezionata l'opzione alto baud rate; anche in questo caso le colonne fanno riferimento a diversi valori di quarzo.

#### Nota:

Per poter utilizzare il modulo EUSART i seguenti bit devono essere così impostati:

- Il bit SPEN<sup>136</sup> (RCSTA bit 7 deve essere impostato ad 1)
- Il registro TRISC bit 7 deve essere impostato ad 1
- Il registro TRISC bit 6 deve essere impostato ad 1

Il modulo EUSART, alla sua abilitazione, imposterà come uscita il bit RC6 (TX) del registro TRISC. Qualora si faccia uso della libreria Microchip, non bisogna preoccuparsi del bit SPEN, il quale viene propriamente settato all'apertura del modulo EUSART.

---

<sup>136</sup> Il bit SPEN abilita il modulo seriale ovvero l'EUSART (*Serial Port Enable*).

Baud Rate bps	SYNC = 0, BRGH = 0, BRG16 = 0											
	Fosc = 40 MHz			Fosc = 20 MHz			Fosc = 10 MHz			Fosc = 8 MHz		
	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG
300	-	-	-	-	-	-	-	-	-	-	-	-
1200	-	-	-	1221	1.73	255	1202	0.16	129	1201	-0.16	103
2400	2441	1.73	255	2404	0.16	129	2404	0.16	64	2403	-0.16	51
9600	9615	0.16	64	9766	1.73	31	9766	1.73	15	9615	-0.16	12
19200	19531	1.73	31	19531	1.73	15	19531	1.73	7	-	-	-
57600	56818	-1.36	10	62500	8.51	4	52083	-9.58	2	-	-	-
115200	125000	8.51	4	104167	-9.58	2	78125	-32.18	1	-	-	-

Baud Rate bps	SYNC = 0, BRGH = 0, BRG16 = 0								
	Fosc = 4 MHz			Fosc = 2 MHz			Fosc = 1 MHz		
	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG
300	300	0.16	207	300	-0.16	103	300	-0.16	51
1200	1202	0.16	51	1201	-0.16	25	1201	-0.16	12
2400	2404	0.16	25	2403	-0.16	12	-	-	-
9600	8929	-6.99	6	-	-	-	-	-	-
19200	20833	8.51	2	-	-	-	-	-	-
57600	62500	8.51	0	-	-	-	-	-	-
115200	62500	-45.75	0	-	-	-	-	-	-

Baud Rate bps	SYNC = 0, BRGH = 1, BRG16 = 0											
	Fosc = 40 MHz			Fosc = 20 MHz			Fosc = 10 MHz			Fosc = 8 MHz		
	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG
300	-	-	-	-	-	-	-	-	-	-	-	-
1200	-	-	-	-	-	-	-	-	-	-	-	-
2400	-	-	-	-	-	-	2441	1.73	255	2403	-0.16	207
9600	9766	1.73	255	9615	0.16	129	9615	0.16	64	9615	-0.16	51
19200	19231	0.16	129	19231	0.16	64	19531	1.73	31	19230	-0.16	25
57600	58140	0.94	42	56818	-1.36	21	56818	-1.36	10	55555	3.55	8
115200	113636	-1.36	21	113636	-1.36	10	125000	8.51	4	-	-	-

Baud Rate bps	SYNC = 0, BRGH = 1 BRG16 = 0								
	Fosc = 4 MHz			Fosc = 2 MHz			Fosc = 1 MHz		
	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG	Actual Rate	Error %	SPBRG
300	-	-	-	-	-	-	300	-0.16	207
1200	1202	0.16	207	1021	-0.16	103	1201	-0.16	51
2400	2404	0.16	103	2403	-0.16	51	2403	-0.16	25
9600	9615	0.16	25	9615	-0.16	12	-	-	-
19200	19231	0.16	12	-	-	-	-	-	-
57600	62500	8.51	3	-	-	-	-	-	-
115200	125000	8.51	1	-	-	-	-	-	-

**Tabella 12:** Valori del registro SPBRG



---

### Esempio:

```
// Quarzo 20MHz
// Formato 8 bit
// 1 bit di stop
// Baud rate 19200bit/s
// Interruzioni disattive

OpenUSART( USART_TX_INT_OFF &
  USART_RX_INT_OFF &
  USART_ASYNC_MODE &
  USART_EIGHT_BIT &
  USART_CONT_RX &
  USART_BRGH_HIGH,
  64 );
```

Si noti che per evitare di scrivere tutte le impostazioni su di una linea si è formattato il tutto su più linee. Per il C questo non è un problema, poiché il compilatore, per capire il termine della riga andrà a cercare il punto e virgola. L'approccio con AND di parametri è molto usato, normalmente lo si implementa per mezzo di una struttura `enum` all'interno della quale si definiscono le costanti da utilizzare.

---

### **char ReadUSART ( void )**

Per mezzo di questa funzione è possibile leggere un byte ricevuto dalla porta seriale; il valore letto ovvero ritornato dalla funzione, è di tipo `char`.

#### **Parametri:**

`void`.

#### **Ritorna:**

Dato letto

### Esempio:

```
char data = 0;

// Aspetta per la ricezione di dati (tecnica del polling)
while ( !DataRdyUSART() );

// Leggo il dato arrivato
data = ReadUSART ();
```

---

### **void WriteUSART ( char data )**

Per mezzo di questa funzione è possibile scrivere un dato in uscita alla porta seriale. Il dato deve essere di tipo `char` quindi di lunghezza non superiore a 8 bit.

#### **Parametri:**

---

**data:** Dato da inviare

**Ritorna:**

void.

**Esempio:**

```
char data = 0x0A;  
  
// Invio il dato  
WriteUSART(data);
```

---

Per ulteriori informazioni sulle altre funzioni disponibili nella libreria `usart.h` si rimanda alla documentazione ufficiale della Microchip. Tra le funzioni non trattate in questo paragrafo ma che possono ritornare utili, si ricorda la famiglia di funzioni `put` e `get` che permettono di scrivere e leggere, singoli caratteri o stringhe, sia da variabili che da stringhe costanti (rom).

Per ulteriori informazioni sulle impostazioni sull'USART si rimanda al datasheet del PIC utilizzato.

## Il codice ASCII

Da quanto appena visto si capisce che la trasmissione per quanto complessa possa essere strutturata avverrà comunque per singoli byte. La trasmissione seriale venne inizialmente utilizzata per i modem e terminali quali la tastiera e mouse. Oggigiorno anche se i vecchi modem sono scomparsi e la tastiera come il mouse sono ormai con interfaccia USB, la trasmissione, il più delle volte, avviene ancora facendo uso di uno standard di caratteri ideato nel 1961. Tale standard venne proposto da l'ingegnere Bob Bemer e successivamente accettato come standard dall'ISO 646<sup>137</sup>. Lo standard fa uso di soli 7 bit dunque anche un semplice `char` (ovvero con segno) è sufficiente per contenere un carattere. In Tabella 13 sono riportati i caratteri ASCII (*American Standard Code for Information Interchange*, ovvero Codice Standard Americano per lo Scambio di Informazioni) .

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0x00	Null	32	0x20	Space	64	0x40	@	96	0x60	`
1	0x01	Start of Heading	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	Start of Text	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	End of Text	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	End of transmit	36	0x24	\$	68	0x44	D	100	0x64	d
5	0x05	Enquiry	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	Acknowledge	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	Audible Bell	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	Backspace	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	Horizontal Tab	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	Line feed	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	Vertical Tab	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	Form Feed	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	Carriage Return	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	Shift Out	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	Shift In	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	Data link Escape	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	Device control 1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	Device control 2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	Device control 3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	Device control 4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	Neg. Acknowledge	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	Synchronous idle	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	End trans. block	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	Cancel	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	End of medium	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	Substitution	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	Escape	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	File Separator	60	0x3C	<	92	0x5C	\	124	0x7C	
29	0x1D	Group Separator	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	Record Separator	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	Unit Separator	63	0x3F	?	95	0x5F	_	127	0x7F	

Tabella 13: Codice ASCII

<sup>137</sup> Fonte Wikipedia, enciclopedia online.

---

E' possibile vedere che i primi 32 caratteri hanno un significato particolare associato al verificarsi di particolari eventi; alcuni di questi eventi saranno utilizzati negli esempi che seguiranno.

Un'altra cosa molto interessante è vedere che i numeri iniziano dal numero 48, dunque per convertire un intero ad una cifra nel suo equivalente ASCII basta sommare 48. Allo stesso modo è possibile vedere che la distanza tra la "a" e la "A" è pari a 0x20, dunque sommando 0x20 è possibile convertire una lettera minuscola nel suo equivalente maiuscolo.

## Esempio di utilizzo dell'EUSART in polling

Come già visto in esempi precedenti, il controllo continuo del verificarsi di un evento viene definito polling. Tale tecnica può essere molto semplice e pratica qualora il PIC non debba svolgere altre mansioni. Nell'esempio che segue si realizza un semplice sistema collegato al PC per mezzo della porta seriale, il testo che verrà scritto con la tastiera verrà visualizzato sul display LCD. Ancora una volta è possibile vedere come grazie all'utilizzo delle librerie il programma risulta piuttosto semplice. Per poter scrivere il testo dal PC al nostro sistema Freedom II si è fatto uso del programma HyperTerminal, presente nel sistema operativo Windows XP<sup>138</sup>. Questo deve essere impostato per operare sulla porta COM seriale dove è connesso Freedom II e deve avere i seguenti parametri:

### Configurazione del terminale

Formato: 8 bit  
Baud Rate: 19200 bit/s  
Stop bit: 1 bit stop  
Bit di parità: 0 bit  
Controllo Flusso: Nessuno

Tale configurazione deve essere utilizzata anche nel caso in cui si faccia uso di altri terminali per il controllo della porta seriale.

```
#include <p18f4550.h>
#include <usart.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilitato il watchdog timer
//LVP = OFF          Disabilitato programmazione LVP
//PBADEN = OFF       Disabilitato gli ingressi analogici

void main (void) {
```

---

<sup>138</sup> Tale programma può essere trovato al menù *Tutti i Programmi* → *Accessori* → *Comunicazioni*. Sfortunatamente tale terminal è stato rimosso dai sistemi operativi successivi a Windows XP. In ogni modo altri terminal o programmi possono essere utilizzati, purché propriamente impostati.

```

// Variabile per salvare il dato di ritorno
unsigned char data = 0;

// Imposto PORTA tutti ingressi
LATA = 0x00;
TRISA = 0xFF;

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0b11111101;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

OpenLCD (20);
BacklightLCD (TURN_ON);

// Configura l'USART
// 8 bit
// 19200 bit/s
// 1 bit stop
// 0 bit parità

OpenUSART( USART_TX_INT_OFF &
USART_RX_INT_OFF &
USART_ASYNC_MODE &
USART_EIGHT_BIT &
USART_CONT_RX &
USART_BRGH_HIGH,
64 );

// Invio la stringa al terminale
putsUSART ("...start writing: ");

// Invio la stringa all'LCD
WriteStringLCD ("...start writing");

// Attendo di ricevere un dato dal PC
while(!DataRdyUSART( ));

ClearLCD ();

// Ciclo infinito
while(1) {

    // Leggo il dato dal buffer di ricezione
    data = ReadUSART();

    WriteCharLCD (data);

    // Invio il carattere al terminale
    WriteUSART (data);

```

```

// Attendo che il dato venga trasmesso
while (BusyUSART());

// Se premo Back Space pulisco lo schermo
if(data == 0x08) {
    ClearLCD ();
}

// Se premo Esc termino l'applicazione
if(data == 0x1B) {
    break;
}

// Attendo di ricevere un dato dal PC
while(!DataRdyUSART( ));
}

// Chiudo l'USART
CloseUSART();

// Ripulisco LCD prima di riscrivere
ClearLCD ();

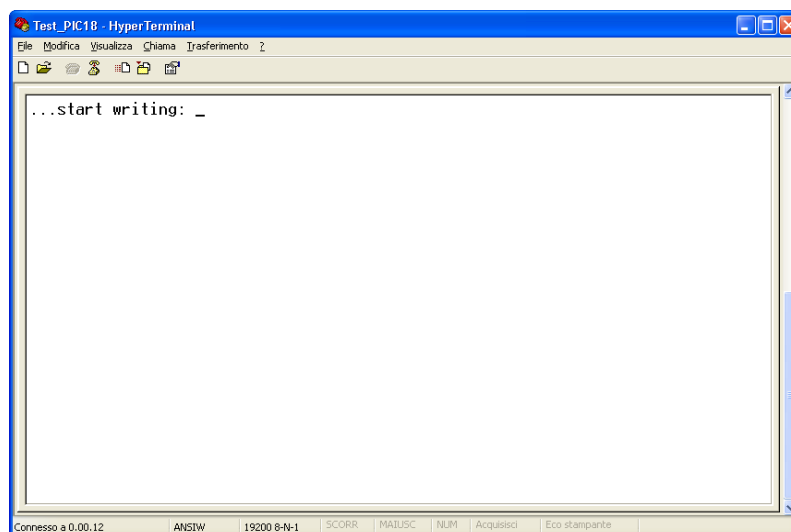
// L'USART è stata chiusa
WriteStringLCD ("RS232 : Closed");

// Ciclo infinito
while (1) {

}
}

```

Arrivati a questo punto del testo è inutile ripetere sempre le stesse cose...è più interessante andare al sodo! Come prima cosa si imposti propriamente la scheda Freedom II e la si colleghi alla porta seriale, si avvi il programma HyperTerminal e successivamente Freedom II propriamente programmata. HyperTerminal, all'avvio di Freedom II mostrerà il messaggio come riportato in Figura 67.



**Figura 67:** *HyperTerminal all'avvio di Freedom II*

Tale messaggio viene anche visualizzato sul display LCD. A questo punto il programma si mette in attesa che l'utente inizi a scrivere sulla tastiera.

```
// Attendo di ricevere un dato dal PC
while(!DataRdyUSART( ));
```

Una volta rilevato un carattere dalla tastiera, viene ripulito lo schermo LCD e si entra in un loop infinito, dal quale si uscirà solo quando il carattere premuto che viene letto è *ESC* (ovvero codice ASCII 0x1B). Entrati nel loop il carattere ricevuto viene letto dal buffer di ricezione e viene scritto sul display LCD e inviato al Terminal, in modo da visualizzare il carattere ricevuto.

Il carattere letto, come detto viene controllato per vedere se è il tasto *ESC*, che fa uscire dal loop infinito o il tasto *Back Space* che permette di cancellare il display<sup>139</sup>. Dopo i controlli si attende nuovamente di ricevere un nuovo carattere.

Si noti che la prima attesa di lettura del carattere è stata posta fuori dal loop `while` in modo da poter scrivere sullo schermo la scritta “...start writing”, per poi cancellarla solo alla ricezione del primo carattere. Se così non si fosse fatto si sarebbe stati costretti a pulire lo schermo all'interno del loop `while`, mettendo però un controllo sul numero di caratteri premuti, in maniera da ripulire il tutto solo alla ricezione del primo carattere.

## Esempio di utilizzo dell'EUSART con interruzione

Ogni volta che il microcontrollore deve eseguire più attività, gestire le periferiche in polling non è sempre l'ottimo. In questo secondo esempio si rivede quanto mostrato in precedenza ma facendo uso delle interruzioni. Concettualmente non è cambiato nulla ma il programma è stato riorganizzato.

```
#include <p18f4550.h>
#include <usart.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

//*****
//                      Gestione Interrupt
//*****

// Prototipo di funzione
void High_Int_Event (void);

#pragma code high_vector=0x08
```

<sup>139</sup> Per cancellare il display non si è fatto uso del tasto *Canc* poiché questo, di default, non viene riconosciuto da HyperTerminal, dunque non viene trasmesso. Per trasmettere il carattere *Canc* è necessario cambiare alcune impostazioni di HyperTerminal.

```

void high_interrupt (void) {

    // Imposta il salto per la gestione dell'interrupt
    _asm GOTO High_Int_Event _endasm
}

#pragma code

#pragma interrupt High_Int_Event

void High_Int_Event (void) {

    // Variabile che conterrà i dati ricevuti
    unsigned char data;

    // Flag per controllare la ricezione del primo carattere
    static char firstTime = 0;

    // Controllo che l'interrupt sia stato generato dall'USART
    if (PIR1bits.RCIF == 1 ) {

        // Controllo la ricezione del primo carattere
        if (firstTime == 0) {
            ClearLCD ();

            // Memorizzo il fatto che ho già pulito il display
            firstTime = 1;
        }

        // Leggo il dato dal buffer di ricezione
        data = ReadUSART();

        WriteCharLCD (data);

        // Invio il carattere al terminale
        WriteUSART (data);

        // Attendo che il dato venga trasmesso
        while (BusyUSART());

        // Se premo Back Space pulisco lo schermo
        if(data == 0x08) {
            ClearLCD ();
        }

        // Se premo Esc termino l'applicazione
        if(data == 0x1B) {

            ClearLCD ();

            // Chiudo l'USART
            CloseUSART();

            // Ripulisco LCD prima di riscrivere
            ClearLCD ();

            // L'USART è stata chiusa
            WriteStringLCD ("RS232 : Closed");

            // Disabilito l'interrupt globale
            INTCONbits.GIE = 0;
        }
    }
}

```



```

    }

    PIR1bits.RCIF = 0;
}
}

//*****
//                               Programma Principale
//*****

void main (void){

    // Variabile per salvare il dato di ritorno
    unsigned char data = 0;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi
    LATC = 0x00;
    TRISC = 0b11111101;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    OpenLCD (20);
    BacklightLCD (TURN_ON);

    // Configura l'USART
    // 8 bit
    // 19200 bit/s
    // 1 bit stop
    // 0 bit parità
    // Interruzione RX abilitata

    OpenUSART( USART_TX_INT_OFF &
    USART_RX_INT_ON &
    USART_ASYNC_MODE &
    USART_EIGHT_BIT &
    USART_CONT_RX &
    USART_BRGH_HIGH,
    64 );

    // Invio la stringa al terminale
    putsUSART ("...start writing: ");

    // Invio la stringa all'LCD
    WriteStringLCD ("...start writing");

    // Abilito modalità compatibile (di default vale già 0)
    RCONbits.IPEN = 0;

```

```

// Abilito l'interrupt globale
INTCONbits.GIE = 1;

// Abilito interrupt per periferiche
INTCONbits.PEIE = 1 ;

// Ciclo infinito
while (1) {

}
}

```

E' possibile vedere che il programma principale, ovvero la funzione `main` ha il compito di inizializzare l'EUSART, scrivere i messaggi ed abilitare le interruzioni in modalità compatibile (ovvero alta priorità).

Alla routine di gestione dell'interrupt viene lasciato il compito di scrivere il carattere sul display e controllare se il tasto premuto è *ESC* o *Back Space*. In particolare alla routine d'interrupt viene lasciato il compito di ripulire lo schermo alla ricezione del primo carattere. Questa volta non essendo stato utilizzato il trucco precedente si è dovuto creare una variabile come `flag`<sup>140</sup>, per memorizzare la ricezione del primo carattere. In questo modo alla ricezione dei successivi caratteri il display non viene ripulito. Si noti che la variabile è stata dichiarata `static` in modo da mantenere il valore assegnato tra una chiamata e l'altra della routine di gestione dell'interrupt.

Si noti inoltre che alla pressione del tasto *ESC* non viene più eseguita l'istruzione `break` bensì il codice per disattivare la periferica e le interruzioni.

Notate come le applicazioni cominciano a divenire sempre più utili. In particolare il limite dell'applicazione è spesso legato alla fantasia umana...spesso senza limiti!

#### **Nota:**

Come visto, in questo esempio si è fatto uso di numeri magici (*magic number*), infatti i pulsanti non sono riconosciuti per mezzo di confronti con costanti ma per mezzo di confronti con valori numerici. Il valore *Back Space* vale in particolare `0x08` che per sua sfortuna è proprio pari al valore del vettore delle interruzioni. Questo significa che se si volesse cambiare il valore `0x08` del pulsante *Back Space* è bene non farlo con Tool automatici (Trova e cambia in...), i quali cambierebbero anche il valore del vettore delle interruzioni. In questo caso non essendo presenti molti numeri `0x08` non è un grave problema, ma il non aver utilizzato una costante sin dal principio potrebbe portare problemi in fase di *Refactoring* del software qualora il numero dovesse comparire in molti punti

<sup>140</sup> Variabili che hanno il compito di memorizzare un determinato stato, quale per esempio acceso/spento vengono spesso chiamate `flag`.

# Capitolo XIII

## Utilizziamo il modulo I2C interno al PIC

Molti microcontrollori, tra cui i PIC18 possiedono oltre alla USART, un modulo I2C per comunicare con periferiche esterne, facendo uso di due soli fili. In questo Capitolo si introdurrà brevemente il bus I2C e le sue applicazioni; successivamente verrà spiegata la modalità per impostare il modulo interno ai PIC18. Il Capitolo è terminato con degli esempi pratici di utilità quotidiana.

### Descrizione dell'hardware e sue applicazioni

Il protocollo I<sup>2</sup>C è uno standard ideato dalla Philips (ora NXP) che ne possiede per altro la proprietà. Venne ideato nel 1980 per superare le difficoltà inerenti all'utilizzo di bus paralleli per la comunicazione tra un'unità di controllo e le varie periferiche.

Le soluzioni che vengono adottate per permettere la comunicazione tra un microcontrollore e le periferiche esterne fanno generalmente uso di una comunicazione parallela. Per tale ragione il bus<sup>141</sup> su cui deve viaggiare l'informazione è costituito da molti fili. Fin quando bisogna collegare una sola periferica al microcontrollore i problemi legati alla presenza di molte linee possono essere tenuti sotto controllo, qualora le periferiche dovessero essere più di una, far giungere il bus ad ogni periferica può diventare un problema.

Un semplice bus ad otto linee comporta comunque la presenza ad ogni integrato di almeno altrettanti pin necessari per la comunicazione, questo significa che le dimensioni dell'integrato vengono ad essere dipendenti dalla dimensione del bus stesso. Ciò comporta che lo stesso PCB (*Print Circuit Board*) sul quale andrà montato l'integrato sarà più grande e quindi più costoso.

Questi problemi vengono interamente superati dal bus I2C, che permette una comunicazione tra periferiche con due sole linee<sup>142</sup>; questa è la ragione per cui non è raro avere integrati con bus I2C a soli otto pin.

Il protocollo I2C è uno standard seriale che a differenza del protocollo RS232<sup>143</sup>, che permette un collegamento punto punto tra due sole periferiche, permette di collegare sullo stesso bus un numero elevato di periferiche, ognuna individuata da un proprio indirizzo.

La possibilità di poter collegare più periferiche sullo stesso bus è permesso anche dal bus RS485 e CAN. Quest'ultimo protocollo è stato ideato per operare in ambienti particolarmente rumorosi e in cui si debba raggiungere un grado di sicurezza nella trasmissione dati particolarmente elevata. Per tali ragioni il bus CAN è ormai accettato come standard in ambito automobilistico, per mettere in comunicazione i vari dispositivi elettronici che sempre più frequentemente vengono installati a bordo.

Un notevole vantaggio dei dispositivi che fanno uso del bus I2C è quello della loro semplicità d'uso. Infatti tutte le regole del protocollo che bisogna rispettare per una corretta comunicazione vengono gestite a livello hardware, dunque il progettista non si deve

<sup>141</sup> Per bus si intende semplicemente un insieme di linee sui cui viaggia un segnale elettrico.

<sup>142</sup> Alle due linee bisogna comunque aggiungere la linea di massa comune.

<sup>143</sup> Si veda il Tutorial "Il protocollo RS232" per maggior chiarimenti sull'argomento.

---

preoccupare di nulla.

Da quando è nato, tale protocollo è stato aggiornato al fine di adeguarlo alle diverse esigenze che il mondo dell'elettronica ha richiesto. Tutte le modifiche apportate sono sempre state compatibili dall'alto verso il basso, ovvero gli integrati che soddisfano gli ultimi standard possono comunicare sempre con gli integrati della generazione precedente.

La prima versione del bus I2C permette di trasmettere fino a 100Kbit/s (modalità standard<sup>144</sup>). Questa velocità è stata portata a 400Kbit/s nelle modifiche apportate nel 1992 (modalità veloce<sup>145</sup>). Nel 1998 la velocità è stata portata fino a 3.4Mbit/s (modalità ad alta velocità<sup>146</sup>). Non necessariamente gli integrati di ultima generazione devono rispettare la modalità ad alta velocità. Tra le periferiche che fanno uso del bus I2C si ricordano: memorie EEPROM, memorie RAM, real time Clock Calendar, LCD, potenziometri digitali, convertitori A/D, sintonizzatori radio, controller per toni DTMF, periferiche generiche per estendere il numero degli ingressi o delle uscite (PCF8574), sensori per la temperatura, controllori audio e molto altro.

Un altro vantaggio che permette di ottenere il bus I2C è quello di poter aggiungere o togliere delle periferiche dal bus senza influenzare il resto del circuito. Questo si traduce in una facile scalabilità verso l'alto del sistema, ovvero si può migliorare un sistema aggiungendo nuove caratteristiche senza dover toccare l'hardware<sup>147</sup>.

Come detto il bus I2C è un bus seriale che necessita di sole due linee nominate SDA (*Serial Data*) e SCL (*Serial Clock*) più la linea di massa. Ambedue le linee sono bidirezionali<sup>148</sup>. La prima è utilizzata per il transito dei dati che sono in formato ad 8 bit, mentre la seconda è utilizzata per trasmettere il segnale di Clock necessario per la sincronizzazione della trasmissione. Il bus I2C permette la connessione di più periferiche su uno stesso bus ma permette la comunicazione tra due soli dispositivi alla volta.

Chi trasmette le informazioni è chiamato trasmettitore mentre chi le riceve è chiamato ricevitore. L'essere il trasmettitore o il ricevitore non è una posizione fissa, ovvero, un trasmettitore può anche divenire ricevitore in una differente fase della trasmissione dati.

In ogni comunicazione è invece fissa la posizione del cosiddetto *Master* (Padrone) e del cosiddetto *Slave* (Schiavo). Il *Master* è il dispositivo che inizia la comunicazione ed è lui a terminarla, lo *Slave* può solo ricevere o trasmettere informazioni su richiesta del Master.

Non tutti i dispositivi possono essere dei *Master* del bus I2C, per esempio una memoria per il mantenimento dei dati non sarà un *Master* del bus, mentre è ragionevole supporre che un microcontrollore lo possa essere<sup>149</sup>.

Su uno stesso bus è inoltre possibile la presenza di più *Master*, ma solo uno alla volta ricoprirà questo ruolo. Se per esempio due microcontrollori iniziano una comunicazione, anche se potenzialmente potrebbero essere ambedue dei *Master*, solo uno lo sarà, in particolare il *Master* sarà quello che ha iniziato la comunicazione, mentre l'altro sarà uno *Slave*. Ogni periferica inserita nel bus I2C possiede un indirizzo che sul bus la individua in modo univoco. Questo indirizzo può essere fissato dal produttore in sede di fabbricazione o parzialmente fissato dal progettista. L'indirizzo è costituito da 7 bit nelle versioni standard o da 10 bit nelle versioni estese. Nel caso di indirizzamento a 7 bit si avrebbe potenzialmente la

---

<sup>144</sup> Standard mode.

<sup>145</sup> Fast speed mode.

<sup>146</sup> High speed mode.

<sup>147</sup> Naturalmente il software dell'unità di controllo dovrà essere cambiato affinché possa riconoscere la nuova periferica. Quanto detto non vale se il software è predisposto per accettare la nuova periferica, la quale può esser dunque inserita senza alcuna modifica né hardware né software.

<sup>148</sup> Dal momento che la linea dati è bidirezionale si ha che il sistema è half-duplex (si veda "Il protocollo RS232" per maggior chiarimenti).

<sup>149</sup> Molti microcontrollori integrano al loro interno l'hardware necessario per la gestione del bus I<sup>2</sup>C sia in modalità *Master* che *Slave*.

---

possibilità di indirizzare 128 periferiche mentre nel caso di 10 bit si avrebbe la possibilità di indirizzare fino a 1024 periferiche. Il numero di periferiche ora citate non sono comunque raggiungibili dal momento che alcuni indirizzi, sono riservati a funzioni speciali. Nel caso in cui l'indirizzo che l'integrato ha all'interno del bus I<sup>2</sup>C venga fissato dall'industria, conduce al fatto che su un bus non potranno essere presenti due integrati dello stesso tipo.

Questa soluzione viene generalmente scelta per per i Real Time Clock Calendar, ovvero per gli orologi, è ragionevole infatti presumere che in un circuito, e in particolare sullo stesso bus, sia presente un solo orologio che mantenga ora e data. L'utilizzo di indirizzi fissi permette di utilizzare dei package ridotti ed ottimizzare le funzionalità offerte dai pin. Per memorie esterne di tipo EEPROM viene invece spesso fornita la possibilità di selezionare l'indirizzo della memoria, impostando alcuni bit dell'indirizzo stesso (normalmente 3).

Per maggiori informazioni sul bus I2C si rimanda al Tutorial “Bus I2C” scaricabile dal sito [www.LaurTec.com](http://www.LaurTec.com).

## Impostare il modulo I2C per un corretto utilizzo

Nonostante si sia detto che il bus I2C sia un modo semplice per comunicare con delle periferiche per mezzo delle due sole linee SDA e SCL, impostare ogni registro del microcontrollore può risultare laborioso, in particolare si può facilmente incorrere in impostazioni errate che possono portare al malfunzionamento del modulo I2C. Per questa ragione l'utilizzo di librerie già collaudate può ritornare particolarmente utile, dunque piuttosto che spiegare ogni registro, per la cui spiegazione si rimanda al datasheet del PIC utilizzato, si spiegherà come impostare il modulo per mezzo della libreria della Microchip. Il file da includere per la libreria è i2c.h. All'interno di questa libreria sono presenti le funzioni per aprire e chiudere il modulo e controllare i vari stati del modulo stesso. Inoltre supporta l'invio di byte e la lettura d'informazione da periferiche esterne, ciononostante in questo paragrafo verranno illustrate solo le funzioni per attivare il modulo ed impostarlo per un corretto utilizzo. Il modo con cui le informazioni verranno lette o scritte è poi lasciato alle librerie specifiche che verranno illustrate nei prossimi paragrafi.

Vediamo ora 2 delle funzioni principali che sono presenti nella libreria i2c.h.

Funzione	Descrizione
void <b>OpenI2Cx</b> (unsigned char, unsigned char )	Apri il modulo I2Cx
void <b>CloseI2Cx</b> (void)	Chiudi il modulo I2Cx

**Tabella 14: Funzioni disponibili nella libreria**

---

### **void OpenI2Cx (unsigned char sync\_mode, unsigned char slew)**

Questa funzione deve essere eseguita prima di utilizzare qualunque libreria che fa utilizzo del modulo I2C, infatti permette di attivare il modulo per un suo corretto utilizzo. Tale funzione potrebbe non essere richiamata solo nel caso in cui la libreria del dispositivo I2C utilizzata, fornisca un'altra funzione per attivare ed impostare il modulo I2C. La x del nome della funzione va sostituita con il numero del modulo utilizzato. Sono infatti presenti PIC con più moduli I2C (fino a 3). In particolare se il PIC possiede un solo modulo I2C il nome della funzione è OpenI2C.

---

**Parametri:**

<b>sync_mode:</b>	SLAVE_7	I2C Modalità <i>Slave</i> con indirizzo a 7 bit.
	SLAVE_10	I2C Modalità <i>Slave</i> con indirizzo a 10 bit.
	MASTER	I2C Modalità <i>Master</i> .
<b>slew:</b>	SLEW_OFF	Ottimizzazione slew-rate disabilitata per lavorare a 100 KHz.
	SLEW_ON	Ottimizzazione slew-rate abilitata per lavorare a 400 KHz.

**Ritorna:**

void.

Oltre ad attivare il modulo è necessario impostare la frequenza di lavoro del modulo stesso. Questo è possibile farlo per mezzo del registro *SSPADD*, il cui valore determina appunto la frequenza operativa che verrà utilizzata qualora il modulo sia impostato come *Master*. L'equazione da usare per il calcolo del valore da inserire in *SSPADD* è:

$$SSPADD = \frac{F_{OSC}}{4 \cdot F_{BUS}} - 1$$

Dove  $F_{OSC}$  rappresenta la frequenza del quarzo mentre  $F_{BUS}$  rappresenta il valore della frequenza che si vuole avere per la trasmissione dei dati sul bus I2C.

---

**void CloseI2Cx (void)**

Questa funzione deve essere eseguita per richiudere il modulo I2C. Tale funzione potrebbe non essere mai richiamata all'interno del programma, qualora non si avesse mai l'esigenza di chiudere il modulo. La x del nome della funzione va sostituita con il numero del modulo utilizzato, sono infatti presenti PIC con più moduli I2C (fino a 3). Se il PIC possiede un solo modulo I2C il nome della funzione è *CloseI2C*.

**Parametri:**

void.

**Ritorna:**

void.

---

Si fa in ultimo presente che generalmente il modulo I2C è mutuamente esclusivo con il modulo SPI. Il modulo SPI è un modulo molto semplice per comunicare con molte altre periferiche di tipologia simile all'I2C. Il modulo I2C lo si preferisce per piccole distanze e frequenze di trasmissioni non elevate. Il modulo SPI viene preferito per lunghe distanze (facendo magari uso di transceiver) e frequenze elevate. Il modulo SPI può essere utilizzato per frequenze dell'ordine di 10MHz senza problemi<sup>150</sup>.

---

<sup>150</sup> A queste frequenze in realtà ci sono problemi! Il layout del PCB risulta molto importante.

---

## Esempio di lettura di una memoria EEPROM I2C

Qualora bisogna salvare in maniera permanente molte informazioni e la memoria EEPROM interna al PIC non è sufficiente, è possibile far uso di memorie EEPROM esterne. Le memorie EEPROM esterne hanno spesso interfacce per bus I2C e SPI. Nel nostro caso si farà riferimento all'interfaccia I2C, per il loro collegamento si rimanda al datasheet delle memoria utilizzata ed in particolare anche al Tutorial “Bus I2C” caricabile dal sito [www.LaurTec.com](http://www.LaurTec.com).

La libreria `i2c.h` della Microchip permette di leggere e scrivere direttamente le memorie EEPROM ma possiede il limite di poter gestire memorie che possiedono un solo byte per l'indirizzamento. Per tale ragione memorie del tipo 24LC32, 24LC64 e così via non possono essere lette o scritte. Come detto la libreria `i2c.h` fornisce però tutte le funzioni generiche per il corretto funzionamento del modulo. Per tale ragione facendo uso della libreria `i2c.h` ho creato una nuova libreria `i2cEEPROM.h` per mezzo della quale è possibile leggere e scrivere in memorie `i2cEEPROM` in cui si hanno due byte per l'indirizzamento della memoria interna, per esempio Freedom II possiede la memoria 24LC32, dunque è richiesta tale libreria. Per il suo corretto funzionamento è necessario includere il file `i2cEEPROM.h` e `i2cEEPROM.lib`.

Dal momento che alcune funzioni fanno utilizzo della libreria `delay.h` è necessario anche includere il file `delay.h` e `delay.lib` scaricabili nel pacchetto libreria di LaurTec. La libreria `i2cEEPROM.h` richiede che il modulo I2C sia stato già propriamente impostato per il corretto funzionamento.

Vediamo ora le funzioni che sono presenti nella libreria `i2cEEPROM.h`.

Funzione	Descrizione
signed char <b>writel2C_EEPROM</b> (unsigned char , int , unsigned char )	Scrive un byte nella cella di memoria indirizzata
signed char <b>writel2C_EEPROM_check</b> (unsigned char , int , unsigned char )	Scrive un byte nella cella di memoria indirizzata e controlla che sia stata scritto propriamente
signed char <b>readl2C_EEPROM</b> ( unsigned char , int , unsigned char )	Legge un byte dalla cella di memoria indirizzata

**Tabella 15: Funzioni disponibili nella libreria**

---

### **signed char writel2C\_EEPROM(unsigned char control, int address, unsigned char data)**

Questa funzione permette di scrivere un byte all'interno della locazione di memoria EEPROM indirizzata.

#### **Parametri:**

**control:** Tale valore rappresenta il parametro di controllo che identifica la memoria. Questo valore lo si trova scritto sul datasheet della memoria utilizzata. Molte memorie `i2c EEPROM` della Microchip possiedono come valore di controllo il valore 10100xxx dove le x rappresentano il valore dei pin d'indirizzo esterni, per esempio Freedom II possiede i pin d'indirizzo della memoria EEPROM collegati a massa, cioè pari a 0; dunque il valore del byte di controllo è 0xA0 (10100000).

**address:** Tale valore rappresenta l'indirizzo della locazione di memoria dove scrivere il byte d'interesse. Tale indirizzo viene scomposto internamente alla funzione in due byte.

---

L'utilizzatore deve accertarsi che l'indirizzo sia sempre nei limiti della memoria utilizzata.

**data:** Tale valore rappresenta il byte che deve essere scritto all'interno della memoria.

**Ritorna:**

- 1: Il byte è stato scritto senza errori (il controllo del valore non è effettuato)
- 1: Errore Bus Collision
- 2: Errore Not Ack Error
- 3: Errore Write Collision

**Esempio:**

```
errore = writeI2C_EEPROM (0xA0, 0x00, 0x55);
```

---

**signed char writeI2C\_EEPROM\_check (unsigned char control, int address, unsigned char data)**

Per ragioni di velocità, la funzione precedente, anche se riporta diverse condizioni di errori, non controlla che il byte sia stato propriamente scritto. In applicazioni in cui bisogna avere un buon livello di robustezza è bene controllare che il byte scritto sia proprio quello voluto, al prezzo di essere più lenti. In particolare il controllo consiste semplicemente nel leggere il byte scritto. Tale controllo risulta anche molto utile in quelle applicazioni in cui si fa uso di molti cicli di scrittura. Si ricorda infatti che le memorie EEPROM hanno un numero limitato, seppur grande, di cicli di scrittura. Dal momento che tale funzione fa utilizzo della libreria `delay.h`<sup>151</sup> è necessario includere il file `delay.h` e `delay.lib` scaricabili nel pacchetto libreria di LaurTec.

**Parametri:**

**control:** Tale valore rappresenta il parametro di controllo che identifica la memoria. Questo valore lo si trova scritto sul datasheet della memoria utilizzata. Molte memorie i2c EEPROM della Microchip possiedono come valore di controllo il valore 10100xxx dove le x rappresentano il valore dei pin d'indirizzo esterni, per esempio Freedom II possiede i pin d'indirizzo della memoria EEPROM collegati a massa, cioè pari a 0; dunque il valore del byte di controllo è 0xA0 (10100000).

**address:** Tale valore rappresenta l'indirizzo della locazione di memoria. Tale indirizzo viene scomposto internamente alla funzione in due byte. L'utilizzatore deve accertarsi che l'indirizzo sia sempre nei limiti della memoria utilizzata.

**data:** Tale valore rappresenta il byte che deve essere scritto all'interno della memoria

**Ritorna:**

---

<sup>151</sup> La funzione `delay` è utilizzata per introdurre l'attesa necessaria al ciclo di scrittura per essere completato; solo dopo alcuni ms (si veda il datasheet) il dato è infatti propriamente scritto.



- 
- 1: Il byte è stato scritto senza errori (il controllo del valore non è effettuato)
  - 1: Errore Bus Collision
  - 2: Errore Not Ack Error
  - 3: Errore Write Collision
  - 4: Il valore scritto e quello voluto differiscono

#### Esempio:

```
errore = writeI2C_EEPROM_check (0xA0, 0x00, 0x55);
```

---

#### **signed char readI2C\_EEPROM (unsigned char control, int address, unsigned char \*data)**

Questa funzione permette di leggere un byte dalla locazione di memoria EEPROM indirizzata.

#### Parametri:

**control:** Tale valore rappresenta il parametro di controllo che identifica la memoria. Questo valore lo si trova scritto sul datasheet della memoria utilizzata. Molte memorie i2c EEPROM della Microchip possiedono come valore di controllo il valore 10100xxx dove le x rappresentano il valore dei pin d'indirizzo esterni, per esempio Freedom II possiede i pin d'indirizzo della memoria EEPROM collegati a massa, cioè pari a 0; dunque il valore del byte di controllo è 0xA0 (10100000).

**address:** Tale valore rappresenta l'indirizzo della locazione di memoria. Tale indirizzo viene scomposto internamente alla funzione in due byte. L'utilizzatore deve accertarsi che l'indirizzo sia sempre nei limiti della memoria utilizzata.

**data:** Tale valore rappresenta l'indirizzo della variabile di tipo `unsigned char`, all'interno della quale verrà scritto il valore letto.

#### Ritorna:

- 1: Il byte è stato scritto senza errori (il controllo del valore non è effettuato)
- 1: Errore Bus Collision
- 2: Errore Not Ack Error
- 3: Errore Write Collision

#### Esempio:

```
unsigned char data;  
signed char errore;  
  
errore = readI2C_EEPROM (0xA0, 0x00, &data);
```

Si noti che la variabile `data` all'interno della quale viene scritto il dato letto viene passata per mezzo dell'operatore `&` che permette di passare l'indirizzo della memoria `data`.

---

Vediamo ora un esempio in cui si scrive un byte nella memoria EEPROM e lo si legge subito dopo per visualizzarlo sui LED.

```
#include <p18f4550.h>
#include <i2cEEPROM.h>
#include <delay.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF          Disabilito il watchdog timer
//LVP = OFF          Disabilito programmazione LVP
//PBADEN = OFF       Disabilito gli ingressi analogici

void main (void){

    // Variabile per salvare il dato di ritorno
    unsigned char data = 0;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi, RC1 come output
    LATC = 0x00;
    TRISC = 0b11111101;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Imposto il quarzo per la libreria delay
    setQuartz (20);

    // Inizializza il modulo I2C a 400KHz @20MHz
    OpenI2C(MASTER, SLEW_ON);
    SSPADD = 12;

    writeI2C_EEPROM (0xA0, 0, 0x35);

    // Aspetto che il dato sia propriamente scritto in EEPROM
    delay_ms (10);

    readI2C_EEPROM (0xA0, 0, &data);

    LATD = data;

    // Ciclo infinito
```

```
while (1) {  
  
}  
}
```

Il programma inizia includendo i file di libreria, si noti che oltre al file di libreria `i2cEEPROM.h` si è incluso anche il file di libreria `delay.h`. Il resto delle impostazioni è piuttosto standard. Come prima istruzione si richiama la seguente funzione:

```
// Imposto il quarzo per la libreria delay  
setQuartz (20);
```

Questo è richiesto per impostare correttamente la funzione `delay.h`. In particolare il valore standard della libreria è già 20MHz, ovvero 20, dunque tale chiamata non è necessaria se il quarzo utilizzato è 20MHz.

Dopo aver impostato la libreria `delay` si imposta il modulo I2C per lavorare a 400 KHz:

```
// Inizializza il modulo I2C a 400KHz @20MHz  
OpenI2C(MASTER, SLEW_ON);  
SSPADD = 12;
```

Solo dopo aver inizializzato il modulo I2C è possibile utilizzare la libreria `i2cEEPROM`.

```
writeI2C_EEPROM (0xA0, 0, 0x35);
```

In questo esempio si ignora il valore di ritorno. Dopo aver scritto il byte 0x35 nell'indirizzo di memoria 0 della EEPROM 24LC32 (il byte controllo di Freedom II è 0xA0) si esegue un ritardo di 10ms. Tale ritardo è necessario per garantire che il byte sia scritto propriamente. Se si eseguisse una lettura prima di tale tempo<sup>152</sup> si rischierebbe di leggere un valore non corretto. Tale tempo non è richiesto tra scritture successive ad indirizzi diversi.

```
// Aspetto che il dato sia propriamente scritto in EEPROM  
delay_ms (10);
```

Una volta attesa la scrittura del byte, viene effettuata la sua lettura.

```
readI2C_EEPROM (0xA0, 0, &data);
```

Si osservi che il valore viene scritto all'interno della variabile `data`, la quale deve essere passata per riferimento, ovvero deve essere passato il suo indirizzo. Si ricorda infatti che i valori passati ad una funzione sono per valore, ovvero la funzione crea delle variabili temporanee all'interno delle quali pone i valori passati. La variazione del contenuto delle variabili temporanee non influenza il valore della variabile passata per valore. Questo è quello che in generale è voluto, ma nel nostro caso dovendo scrivere il valore nella variabile passata è necessario sapere il suo indirizzo reale e non costruire una variabile temporanea. Per fare questo la funzione accetta un puntatore al tipo di variabile che si vuole modificare (nel nostro caso `unsigned char`). Per passare l'indirizzo della variabile si fa uso dell'operatore `&` prima della variabile, da non confondere con l'operatore *and*. Nel caso si volesse passare l'indirizzo di un Array, come detto, si sarebbe fatto uso del nome dell'Array senza parentesi quadre e non dell'operatore `&`. Tale apparente complicazione è stata utilizzata poiché il valore di ritorno è

<sup>152</sup> Il datasheet della memoria 24LC32 mette come valore limite 5ms. Si è scelto di usare 10ms quale forma di garanzia per un corretto funzionamento.

utilizzato per riportare eventuali errori.

...in ultimo, il valore data viene scritto sui LED.

## Esempio di utilizzo di un Clock Calendar I2C

Un'altra periferica frequentemente utilizzata sul bus I2C è il Real Time Clock Calendar. La scheda di sviluppo Freedom II possiede il Clock Calendar della NXP (Philips) PCF8563. Viste le numerose impostazioni che la periferica richiede, ho deciso di scrivere direttamente una libreria dedicata, per utilizzare la libreria bisogna includere il file PCF8563.h e il file PCF8563.lib.

In Tabella 11 sono riportate le funzioni presenti all'interno della libreria PCF8563, come visibile dal numero la libreria offre molta flessibilità. Si noti che in questo caso si è preferito utilizzare una nomenclatura delle funzioni simile a quella utilizzata nell'ambito della programmazione Linux, la ragione è legata al fatto che sono presenti molte parole nel nome della funzione stessa. Se da un lato il nome diviene piuttosto lungo, dall'altro bisogna riconoscere che il nome della funzione è piuttosto autoesplicativo.

Funzione	Descrizione
signed char <b>set_seconds_RTCC</b> (unsigned char);	Imposta i secondi dell'orario corrente.
unsigned char <b>get_seconds_RTCC</b> (void);	Ritorna i secondi dell'orario corrente.
signed char <b>set_minutes_RTCC</b> (unsigned char);	Imposta i minuti dell'orario corrente.
unsigned char <b>get_minutes_RTCC</b> (void);	Ritorna i minuti dell'orario corrente.
signed char <b>set_hours_RTCC</b> (unsigned char);	Imposta le ore dell'orario corrente.
unsigned char <b>get_hours_RTCC</b> (void);	Ritorna le ore dell'orario corrente.
unsigned char* <b>get_time_seconds_RTCC</b> (void);	Ritorna l'orario corrente con i secondi, formato HH:MM.ss.
unsigned char* <b>get_time_RTCC</b> (void);	Ritorna l'orario corrente senza i secondi, formato HH:MM.
signed char <b>set_days_RTCC</b> (unsigned char);	Imposta il giorno della data corrente.
unsigned char <b>get_days_RTCC</b> (void);	Ritorna il giorno della data corrente.
signed char <b>set_day_of_the_week_RTCC</b> (unsigned char );	Imposta il giorno della settimana.
signed char <b>get_day_of_the_week_RTCC</b> (void);	Ritorna il giorno della settimana.
signed char <b>set_months_RTCC</b> (unsigned char);	Imposta il mese della data corrente.
unsigned char <b>get_months_RTCC</b> (void);	Ritorna il mese della data corrente.
signed char <b>set_years_RTCC</b> (unsigned char);	Imposta l'anno della data corrente.
unsigned char <b>get_years_RTCC</b> (void);	Ritorna l'anno della data corrente.
unsigned char* <b>get_date_RTCC</b> (void);	Ritorna la data corrente, formato DD/MM/YY.
signed char <b>set_minutes_alarm_RTCC</b> (unsigned char, unsigned char);	Imposta i minuti dell'allarme.
signed char <b>set_hours_alarm_RTCC</b> (unsigned char, unsigned char);	Imposta le ore dell'allarme.
signed char <b>set_days_alarm_RTCC</b> (unsigned char, unsigned char);	Imposta il giorno per l'allarme.
signed char <b>set_day_of_the_week_alarm_RTCC</b> (unsigned char, unsigned char );	Imposta il giorno della settimana per l'allarme.

Funzione	Descrizione
signed char <b>enable_alarm_interrupt_RTCC</b> (void);	Abilita l'interrupt per la sveglia (allarme).
signed char <b>disable_alarm_interrupt_RTCC</b> (void);	Disabilita l'interrupt per sveglia (allarme).
unsigned char <b>is_alarm_ON_RTCC</b> (void);	Controlla se l'allarme è attivo.
signed char <b>increment_minutes_RTCC</b> (void);	Incrementa i minuti dell'orario corrente.
signed char <b>increment_hours_RTCC</b> (void);	Incrementa le ore dell'orario corrente.
signed char <b>increment_years_RTCC</b> (void);	Incrementa gli anni della data corrente.
signed char <b>increment_months_RTCC</b> (void);	Incrementa i mesi della data corrente.
signed char <b>increment_days_RTCC</b> (void);	Incrementa i giorni della data corrente.

**Tabella 16:** Funzioni disponibili nella libreria

Piuttosto che vedere il funzionamento di ogni funzione, per la cui descrizione si rimanda al file header della libreria stessa, si descrivono in breve alcune delle caratteristiche generali.

Ogni funzione set e comunque ogni funzione che ritorna una variabile `signed char`, ritorna il seguente codice di errore:

#### Ritorna:

- 1: Il byte è stato scritto senza errori (il controllo del valore non è effettuato)
- 1: Errore Bus Collision
- 2: Errore Not Ack Error
- 3: Errore Write Collision

Il formato dei secondi, minuti, ore, giorno, mese, anno sono BCD, ovvero per scrivere il giorno 22 non si può scrivere:

```
set_days_RTCC (22);
```

Facendo però uso del formato esadecimale si può scrivere direttamente:

```
set_days_RTCC (0x22);
```

Le funzioni `get_time_RTCC`, `get_time_seconds_RTCC` e `get_date_RTCC`, ritornano puntatori a una stringa di caratteri ASCII contenente l'orario. La stringa può essere direttamente stampata su di un display LCD alfanumerico.

Vediamo ora un esempio in cui si realizza un orologio con data. Il sistema permette inoltre di cambiare orario e data per mezzo dei pulsanti BT1-BT4.

```
#include <p18f4550.h>
#include <portb.h>
#include <PCF8563.h>

#define LCD_DEFAULT
#include <LCD_44780.h>

#pragma config FOSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
```

```

#pragma config PBADEN = OFF

//OSC = HS          Impostato per lavorare ad alta frequenza
//WDT = OFF         Disabilitato il watchdog timer
//LVP = OFF         Disabilitato programmazione LVP
//PBADEN = OFF      Disabilitato gli ingressi analogici

#define BT1 0b11100000
#define BT2 0b11010000
#define BT3 0b10110000
#define BT4 0b01110000

/*****
// Prototipo di funzione
*****/
void High_Int_Event (void);

//Interrupt vector per modalità compatibile
#pragma code high_interrupt_vector = 0x08

void high_interrupt (void) {

    // Salto per la gestione dell'interrupt
    __asm GOTO High_Int_Event __endasm
}

#pragma code

/*****
// Gestione Interrupt
*****/
#pragma interrupt High_Int_Event

// Funzione per la gestione dell'interruzione
void High_Int_Event (void) {

    // Indice per il ciclo di pausa
    int i;

    unsigned char button;

    // Controllo che l'interrupt sia stato generato da PORTB
    if (INTCONbits.RBIF == 1 ) {

        //pausa filtraggio spike
        delay_ms (30);

        button = PORTB;
        button = button & 0xF0;

        // Controllo del tasto premuto

        switch(button) {

            case BT1 & BT3:
                increment_minutes_RTCC ();
                break;

            case BT1 & BT4:
                increment_hours_RTCC ();
                break;

```

```

        case BT2:
            increment_years_RTCC ();
            break;
        case BT3:
            increment_months_RTCC ();
            break;
        case BT4:
            increment_days_RTCC ();
            break;
    }

    // Resetto il flag d'interrupt per permettere nuove interruzioni
    INTCONbits.RBIF = 0;
}
}

/*****
//          Programma Principale
*****/

void main (void){

    // Variabile per salvare il dato di ritorno
    unsigned char data = 0;

    // Imposto PORTA tutti ingressi
    LATA = 0x00;
    TRISA = 0xFF;

    // Imposto PORTB tutti ingressi
    LATB = 0x00;
    TRISB = 0xFF;

    // Imposto PORTC tutti ingressi, RC1 come output
    LATC = 0x00;
    TRISC = 0b11111100;

    // Imposto PORTD tutte uscite
    LATD = 0x00;
    TRISD = 0x00;

    // Imposto PORTE tutti ingressi
    LATE = 0x00;
    TRISE = 0xFF;

    // Inizializza il modulo I2C a 400KHz @20MHz
    OpenI2C(MASTER, SLEW_ON);
    SSPADD = 12;

    OpenLCD (20);
    BacklightLCD (TURN_ON);

    // Inizializzo la data
    set_days_RTCC (0x30);
    set_months_RTCC (0x12);
    set_years_RTCC (0x08);

    // Inizializzo l'ora
    set_hours_RTCC (0x02);
    set_minutes_RTCC (0x56);
    set_seconds_RTCC (0x33);

```

```

// Abilita i resistori di pull-up sulla PORTB
EnablePullups();

// Abilito le interruzioni su PORTB
INTCONbits.RBIE = 1;

// Abilito modalità compatibile (di default vale già 0)
RCONbits.IPEN = 0;

// Abilito l'interrupt globale
INTCONbits.GIE = 1;

// Abilito l'interrupt periferiche
INTCONbits.PEIE = 1 ;

// Ciclo infinito
while (1) {

    WriteStringLCD ("Time :      ");

    WriteVarLCD (get_time_RTCC ());

    Line2LCD ();
    WriteStringLCD ("Date :  ");

    WriteVarLCD (get_date_RTCC ());

    HomeLCD ();

}
}

```

Questo rappresenta un'altra volta un programma che sarebbe bene dividere in più file, in modo da poter mantenere la chiarezza necessaria. Si noti come risulta facile cambiare la data e l'orario grazie alla struttura `switch` e delle funzioni di libreria.

L'orologio funziona nel seguente modo:

Una volta avviato viene visualizzata una data ed un orario di default (30/12/2008<sup>153</sup>). Questi possono essere cambiati per mezzo dei pulsanti BT1-BT4.

- ✓ Premendo BT2 si incrementa l'anno.
- ✓ Premendo BT3 si incrementa il mese.
- ✓ Premendo BT4 si incrementa il giorno.

Tenendo premuto BT1:

- ✓ Premendo BT3 si incrementano i minuti.
- ✓ Premendo BT4 si incrementano le ore.

Si osservi come i commenti e il nome delle funzioni aiutino alla comprensione del programma...che non verrà ora spiegato!

#### **Nota:**

Per poter far funzionare il programma correttamente può essere necessario rimuovere il programmatore.

<sup>153</sup> Compleanno del mio primo nipotino...



## Bibliografia

### Internet Link

- [1] [www.LaurTec.com](http://www.LaurTec.com) : sito dove scaricare il testo originale del libro “C18 Step by Step” e la documentazione tecnica di Freedom II.
- [2] [www.microchip.com](http://www.microchip.com) : sito ufficiale della Microchip, dove poter scaricare i datasheet dei PIC e i software descritti nel testo.
- [3] [www.zilog.com](http://www.zilog.com) : sito della Zilog, società leader nella produzione di microcontrollori.
- [4] [www.ti.com](http://www.ti.com) : sito della Texas Instrument, società leader nella produzione di microcontrollori e componenti analogici/digitali. .
- [5] [www.analog.com](http://www.analog.com) : sito della Analog Devices, società leader nella produzione di microcontrollori e componenti analogici/digitali.
- [6] [www.maxim-ic.com](http://www.maxim-ic.com) : sito della Maxim IC, società leader nella produzione di microcontrollori e componenti analogici/digitali.
- [7] [www.atmel.com](http://www.atmel.com) : sito della Atmel, società leader nella produzione di microcontrollori.
- [8] [www.st.com](http://www.st.com) : sito della ST Microelectronics, società leader nella produzione di microcontrollori e componenti analogici/digitali.
- [9] [www.renesas.com](http://www.renesas.com) : sito della Renesas, società leader nella produzione di microcontrollori.
- [10] [www.nxp.com](http://www.nxp.com) : sito della NXP, società leader nella produzione di microcontrollori e componenti analogici/digitali. .
- [11] [www.nxp.com/acrobat\\_download/applicationnotes/AN10216\\_1.pdf](http://www.nxp.com/acrobat_download/applicationnotes/AN10216_1.pdf) : documento sulle specifiche I2C.
- [12] [www.national.com](http://www.national.com) : sito dove scaricare il datasheet del sensore di temperatura LM35.

### Libri

- [13] The Data Conversion Handbook (Analog Devices, ed. Elsevier)
- [14] Op Amp Application Handbook (Analog Devices, ed. Elsevier)
- [15] Understanding Digital Signal Processing (R. G. Lyons, ed. Prentice Hall)
- [16] Teoria dei Segnali (Marco Luise, Giorgio M. Vitetta, ed. McGraw-Hill)
- [17] Linux for Embedded and Real Time Applications (Doug Abbott, ed. Newnes)
- [18] Building Embedded Linux Systems (karim, Jon, Gilad & Philippe, ed. O'Reilly)

## Indice Alfabetico

<b>1</b>			
	18f4550_g.lkr.....	49, 59	
<b>2</b>			
	24LC32.....	255	
	24MHz.....	34	
<b>4</b>			
	48MHz.....	34	
<b>6</b>			
	6MHz.....	34	
<b>8</b>			
	8051.....	9, 14	
	8052.....	14	
	80x86.....	13	
<b>A</b>			
	accumulatore.....	25, 101	
	ACQT0.....	218	
	ACQT2.....	218	
	ADC.....	44, 144, 208	
	adc.h.....	218	
	ADCON0.....	212, 218	
	ADCON1.....	44, 46, 212, 215, 218	
	ADCON2.....	212, 215, 218	
	Add SFR.....	82	
	Add Symbol.....	82	
	ADDRESH.....	212, 216, 218	
	ADDRESL.....	212, 216, 218	
	ADIE.....	218	
	ADIF.....	218	
	ADRESH.....	222	
	ADRESL.....	222	
	Advanced RISC Machine.....	13	
	Altera.....	12	
	ALU.....	12, 25	
	AMD.....	9	
	Analisi temporale.....	84	
	Analog Devices.....	14	
	Analog to Digital Converter.....	208	
	ANSI C.....	87, 95	
	Architettura base.....	11	
	architettura base dei PIC18.....	15	
	Architettura media.....	11	
	Architettura media potenziata.....	11	
	Architettura PIC18.....	11	
	Arithmetic Logic Unit.....	12	
	ARM.....	12 e seg.	
	Array.....	93	
	Array di caratteri.....	97, 199	
	ASCII.....	65, 242	
	assembler.....	48	
	AT91SAM.....	13	
	ATMEL.....	13	
	Atom.....	10	
	auto-programmazione.....	20	
	AVR.....	13	
	AVR32.....	13	
<b>B</b>			
	banchi di memoria.....	24	
	bank.....	24	
	Base Architecture.....	11	
	BCF.....	27	
	bin.....	49	
	Bit Clear f.....	27	
	bootloader.....	20	
	BOR.....	39	
	BOREN1:BOREN0.....	40	
	BORV1:BORV0.....	40	
	breadboard.....	54	
	break.....	124, 131	
	Breakpoint.....	75, 80	
	BRGH.....	238	
	BSF.....	27	
	BSR.....	25, 145	
	BSR3:BSR0.....	25	
	buffer overflow.....	94	
	Build Library target.....	162	
	Build Normal target.....	163	
	Build Options.....	60, 62, 156, 164	
	Build Options,.....	162	
	BUILD SUCCEEDED.....	64	
	bus CAN.....	250	
	bus I2C.....	250	
	Busy.....	80	
<b>C</b>			
	C Academic Version.....	47	
	C for PIC18.....	48	
	c018i.c.....	78	
	C2000.....	14	
	cache.....	16	
	CAN.....	250	
	cane da guardia.....	40	
	case.....	124	
	case sensitive.....	89	
	casting.....	104, 222	
	CCP1.....	189	
	CCP2.....	189	
	Central Processing Unit.....	12, 15	
	CFGFS.....	26	
	char.....	90, 95, 128	

ciclo infinito.....	116	documentazione.....	50
circuiteria di Reset.....	38	DOS.....	74
CISC.....	16	DSC.....	11
clock secondario.....	34	DSP.....	11, 13
CLRWDT.....	40	dsPIC.....	11
code.....	148	dsPIC30.....	11
Complex Instruction Set Computer.....	16	dsPIC33.....	11
Complex Programmable Logic Device.....	12	duty cycle.....	180, 183, 186, 189
comunicazione parallela.....	233	<b>E</b>	
CONFIG.....	31, 69	EC.....	32
CONFIG1H.....	31, 34, 36	ECCP.....	183
CONFIG1L.....	31	ECIO.....	32
CONFIG2L.....	40	ECPIO.....	32
CONFIG4L.....	41	ECPLL.....	32
Configuration Bits set in code.....	70	EEADR.....	26
Configuration Register.....	184	EECON1.....	26
controllo dei motori.....	45	EECON2.....	26
controllore HD44780.....	190	EEDATA.....	26
corrente di leakage.....	216	EEPGD.....	26
cos(x).....	100	EEPROM.....	26
costante.....	97	EEPROM Memory.....	20
CPLD.....	12	Effective number of bit.....	211
CPP.....	183	efficienza.....	34
CPU.....	15	Electrically Erasable Programmable Read Only Memory.....	26
CPUDIV.....	34	EnablePullups();.....	119
<b>D</b>		Enhanced Midrange Architecture.....	11
DAC.....	183	Enhanced PWM.....	45
data logger.....	52	ENOB.....	211
Data Memory.....	15, 20	enum.....	98
Debug.....	62	esponente.....	92
DEBUG.....	22	EUSART.....	45, 233
debugger.....	52	Execute.....	23
Debugger → Settings.....	84	<b>F</b>	
default.....	125	Fail Safe Clock Monitor.....	34
delay.h.....	255	famiglia Midrange.....	16
differenziale.....	211	FCMEN.....	34
Digital Signal Controller.....	11	fetch.....	21
Digital Signal Processing.....	13	FFT.....	11
Digital Signal Processor.....	11	Field Programmable Gate Array.....	12
Digital to Analog Converter.....	183	file .lib.....	154
Direct Memory Access.....	11, 16	file header .h.....	154
Directory.....	60	filtro antirimbalo.....	120
direttiva #define.....	156	Finite Impulse Response.....	13
direttiva #endif.....	157	FIR.....	13
direttiva #error.....	158	floating point.....	92
direttiva #ifndef.....	157	for(...).....	112
direttiva #include.....	156	Fosc/4.....	32
direttiva #warning.....	158	FOSC3:FOSC0.....	36
Disassembly Listing.....	91, 101	FPBW.....	211
display alfanumerici.....	190	FPGA.....	12
DMA.....	11, 16	frequenza di campionamento.....	211
doc.....	50		

Full Bridge.....	184	Intel HEX.....	65
Full Power Bandwidth.....	211	Intellectual Propriety.....	26
full-duplex.....	234	interrupt.....	144
<b>G</b>		Interrupt Vector.....	22
General Purpose Register.....	25	interruzione a bassa priorità.....	144
General software library.....	50	interruzione ad alta priorità.....	144
GIE.....	146, 218	INTHS.....	32
giustificazione a destra.....	216	INTIO.....	33
giustificazione a sinistra.....	216	INTOSC.....	36
GO/DONE.....	218	INTRC.....	34, 40
GOTO.....	148	INTXT.....	33
GPR.....	25	IP.....	26
<b>H</b>		IPEN.....	146
h.....	50	IR.....	23
Half Bridge.....	184	IRCF.....	40
half-duplex.....	234	ISR.....	144
hard Real Time.....	166	Istruction Register.....	23
Harvard.....	15	istruzione break.....	130
HI-TECH.....	48	istruzione condizionale switch.....	124
High Priority Interrupt Vector.....	22	istruzione if (...). .....	117
high speed.....	34	Istruzione RESET.....	39, 41
High_Int_Event.....	148	istruzione return (...). .....	136
high_vector.....	148	istruzione while (...). .....	129
Hitachi.....	190	Istruzioni bloccanti.....	74
hlpC18Lib.....	50	<b>K</b>	
hlpC18ug.....	50	kernel Linux.....	74
hlpPIC18ConfigSet.....	50	<b>L</b>	
hlpPIC18ConfigSet.chm.....	69	LAN.....	233
HS.....	31, 70	LAT.....	43
HSPLL.....	32	latency.....	145, 176
<b>I</b>		LCD.....	190
I/O.....	42	LCD_DEFAULT.....	193
I2C.....	26, 250, 252	LDO.....	183
i2c.h.....	253, 255	Le direttive.....	156
i2cEEPROM.h.....	255	left value.....	106
ICD 2.....	53, 79	lettura di un pulsante.....	118
ICD 3.....	53	lib.....	51
ICD2.....	66	Library Search Path.....	60
IDE.....	48	linker.....	49, 65
IDLEN.....	35	Linker Script.....	59
IIR.....	13	Linker-Script Search Path.....	60
INCKO.....	33	Linux.....	10
Include Search Path.....	60, 156	Linux embedded.....	14
Infinite Impulse Response.....	13	LKR.....	49
inizializzazione delle variabili.....	93	log(x).....	100
Input/Output.....	42	long.....	90
int.....	95	Low Drop Out.....	183
int.....	90	Low Priority Interrupt Vector.....	22
INTCON.....	146, 151 e seg., 170	low speed.....	34
INTCON2.....	45	Low Voltage Differential Signal.....	233
Integrated Development Environment.....	48	lvalue.....	106
Intel.....	9	LVDS.....	233

## M

macro.....	97
magic number.....	97, 249
main.....	71
mantissa.....	92
Master.....	252
Master Clear Reset.....	39
Math library.....	50
math.h.....	100
Maxim.....	14
Maxim IC.....	14
MCC18.....	49
MCLR.....	39, 53
MCU.....	11
Memoria Dati.....	20
Memoria EEPROM.....	20
memoria flash.....	20
memoria OTP.....	20
Memoria Programma.....	20
Memory Managment Unit.....	10
Memory Usage Gauge.....	66
menù Configure → Configuration bits.....	69
menù Debugger → Select Tool.....	76, 79
menù Debugger → Settings.....	79
menù Debugger → StopWatch.....	84
menù File → New.....	59
menù Programmer → Select Programmer.....	66
menù Project.....	55
menù Project → Build All.....	63
menù View → Memory Usage Gauge.....	66
menù View → Watch.....	82
Midrange Architecture.....	11
mikroElektronika.....	48
MIPS.....	13
MMU.....	10
modalità Capture.....	184
modalità Compare.....	184
modalità compatibile.....	146
modalità Debug.....	75
modalità estesa.....	16
Modalità Idle.....	35
Modalità PRI_IDL.....	36
Modalità PRI_RUN.....	36
Modalità RC_IDL.....	36
Modalità RC_RUN.....	36
modalità Release.....	75
Modalità Run.....	35
Modalità SEC_IDL.....	36
Modalità SEC_RUN.....	36
Modalità Sleep.....	35, 37
moltiplicatore 8x8.....	19
MOS.....	210

MOVF.....	145
MOVFF.....	145
MOVWF.....	145
mplab.....	163
MPLAB.....	47
MPLAB SIM.....	76
MPLAB-C18-Getting-Started.....	50
mplib.....	162 e seg.
MPLIB.....	163
mplink.....	163
MSP430.....	14
MUX.....	31

## N

NXP.....	14, 250, 260
----------	--------------

## O

operatore ++.....	100
operatore binario AND.....	108
operatore binario OR.....	108
operatore binario XOR.....	108
operatore complemento a 1.....	108
operatore di decremento.....	100
operatore divisione.....	100
operatore logico AND.....	108
operatore logico di uguaglianza.....	108
operatore logico diverso.....	108
operatore logico maggiore.....	108
operatore logico maggiore o uguale.....	108
operatore logico minore.....	108
operatore logico minore o uguale.....	108
operatore logico OR.....	108
operatore moltiplicazione.....	100
operatore somma.....	100
operatore sottrazione.....	100
operatori bitwise.....	108
operatori logici.....	108
operatori matematici.....	100
operazioni binarie.....	25
Osc/Trace.....	84
OSCCON.....	31, 34 e seg.
overflow.....	25, 107

## P

p18cxxx.h.....	160
p18f4550.h.....	50, 69
PAL.....	12
parallelismo.....	19
PBADEN.....	45, 70, 215
PC.....	21, 144
PCB.....	250
PCF8563.....	260
PCF8563.lib.....	260
PCF8574.....	252
PEIE.....	146

Pentium II®.....	10	pull-up.....	45, 119
percorso radice.....	49	Pulse Width Modulation.....	180
percorso standard d'installazione.....	49	puntatore.....	202
Phase Lock Loop.....	31	puntatore di tipo char.....	205
Philips.....	14, 250	PWM.....	45, 180
PIC10.....	11	<b>Q</b>	
PIC10F.....	48	quanto.....	210
PIC12.....	11	QVGA.....	11
PIC12F1.....	11	<b>R</b>	
PIC16.....	11, 16, 146	RA6.....	32 e seg.
PIC16F1.....	11	RAM.....	16, 24, 140, 206, 233
PIC18 Architecture.....	11	RAM.....	90
PIC24.....	11	Random Access Memory.....	24
PIC32.....	12, 48	RB4-RB7.....	144
PICKIT 2.....	52, 75	RB6.....	53
PICKIT 3.....	52	RB7.....	53
PICKIT 3,.....	75	RBIE.....	151
pipeline.....	23	RBIF.....	152
PLD.....	12	RBPU.....	45
PLL.....	31	RC_IDL.....	35
polling.....	28, 144, 218	RC_RUN.....	35
POR.....	39, 41, 214	RCON.....	39
PORT.....	43	RCSTA.....	238
PORTA.....	44, 215	ReadADC ().....	226
PORTB.....	44, 70, 144, 215	Real Time.....	166
PORTC.....	45	Real Time Clock Calendar.....	168, 260
PORTD.....	45, 73	Real Time Operative System.....	10, 23
PORTDbits.....	51	Reduced Instruction Set Computer.....	15
PORTE.....	46	Refactoring.....	249
postscaler.....	31	registri shadow.....	145
potenza dissipata.....	34	Registro di Stato.....	25
power cycle.....	36	registro speciale PC.....	21
Power Cycle.....	28	registro speciale W.....	25
Power On Reset.....	39, 214	Renesas.....	14
Power On Reset.....	39	Reset.....	144
Power-Up.....	28	RESET.....	41
prescaler.....	31, 169, 176	Reset and Connect to ICD.....	79
Prescaler.....	168, 185	Reset Hardware.....	41
PRI_IDL.....	35	Reset Vector.....	22, 38
PRI_RUN.....	35	resistori di pull-up.....	45, 151
Print Circuit Board.....	250	resto divisione tra interi.....	100
Program Counter.....	21, 38, 136	return ().....	136
Program Memory.....	15, 20	RISC.....	13, 15
Programmable Brown-out Reset.....	39	rom.....	207
Programmable Brown-out Reset (BOR).....	40	RS232.....	233
programmatore.....	20, 52	RTOS.....	10, 23
Programmer Logic Analyzer.....	52	Run.....	80
Programming Practices.....	51	Run to Cursor.....	80
Project wizard.....	55	<b>S</b>	
protocollo I2C.....	250	Sample and Hold.....	210
prototipo di funzione.....	137	save.....	178
pull-down.....	45, 119	scheduling dei processi.....	166

SCL.....	252	Stack vuoto.....	39
scope di variabili.....	140	static.....	142
SCS1:SCS0.....	34 e seg.	STATUS.....	145
SDA.....	252	Status Register.....	25
SEC_IDL.....	35	Step Into.....	80
SEC_RUN.....	35	STKFUL.....	41
segnale analogico.....	210	STKPTR.....	22, 41
segnale tempo continuo.....	210	STKUNF.....	41
Self-programming under software control.....	20	StopWatch.....	85
sen(x).....	100	string.h.....	205
Serial Clock.....	252	stringa.....	97, 199
Serial Data.....	252	stringhe.....	204
Serial Port Enable.....	238	struct.....	95
SFDR.....	211	struttura di von Neumann.....	16
SFRs.....	25	struttura enum.....	98
SH.....	210	struttura Harvard.....	15
shift.....	226	STVREN.....	41
shift a destra.....	104, 108	SuperH.....	14
shift a sinistra.....	104, 108	switch.....	124
short.....	90	Synch.....	85
short long.....	90	<b>T</b>	
Show Directories for.....	60	T0CON.....	169
Signal to Noise And Distortion.....	211	target.....	53
Signal to Noise Ratio.....	211	Target Board.....	67
signed char.....	90	tecniche di predizione dei salti.....	16
simplex.....	234	Texas Instrument.....	14
SINAD.....	211	TI.....	14
single ended.....	211	time budget.....	166
sistema real time.....	145	timers.h.....	176 e seg., 186
skew.....	233	tipo string.....	204
Slave.....	252	TMR0H.....	178
Sleep.....	35	TMR0IE.....	170
SLEEP.....	35	TMR0IF.....	170
SNR.....	211	TMR0IP.....	170
soft Real Time.....	166	TMR0L.....	178
Software peripheral library.....	50	TMR2.....	185
spbrg.....	238	tool per il Debug.....	75
Special Function Registers.....	25	Tool Suite C18.....	55
SPEN.....	238	trasmissione full-duplex.....	234
SPI.....	26	trasmissione half-duplex.....	234
spike.....	152	trasmissione parallela.....	233
Spurious Free Dynamic Range.....	211	trasmissione seriale.....	233
ST.....	14	trasmissione simplex.....	234
ST Microelectronics.....	14	trasmissione sincrona.....	234
ST10.....	14	trasmissioni asincrone.....	234
ST6.....	14	Trigger di Schmitt.....	46
ST7.....	14	TRIS.....	42
Stack.....	22	TRISD.....	194
Stack Memory.....	136	typedef struct.....	95
Stack pieno.....	39	<b>U</b>	
Stack Pointer.....	22, 136	UART.....	233
Stack Reset.....	41	unsigned char.....	90

unsigned int.....	90	XT.....	31
unsigned long.....	90	XTPLL.....	31
unsigned short.....	90	<b>Z</b>	
unsigned short long.....	90	Z80.....	9, 14
USART.....	144, 233	Zero.....	85
usart.h.....	235, 241	Zilog.....	14
USB.....	31, 45		
<b>V</b>		Logic Unit.....	25
Value.....	82	—	
variabile static.....	141	_asm.....	148
virus.....	94	_endasm.....	148
Visibilità delle variabili.....	140	.	
void.....	71, 136, 148	.lkr.....	49
volatile.....	152	<b>#</b>	
von Neumann.....	16	#define.....	97 e segg., 124, 126, 156
<b>W</b>		#endif.....	157 e seg., 160
Watch.....	82	#error.....	158
Watchdog Timer.....	35, 39	#ifndef.....	157
Watchdog Timer (WDT).....	40	#include.....	69, 139, 154, 156
WDT.....	39, 70	#include “nome_file”.....	139
Wikipedia.....	66	#include <nome_file>.....	139
Windows.....	10, 74	#include <pwm.h>.....	184
WR.....	28	#include <string.h>.....	97
WREG.....	145	#pragma.....	69, 148, 178
WREN.....	28	#pragma code.....	147
WRERR.....	28	#pragma interrupt.....	148
<b>X</b>		#pragma interruptlow.....	148
Xilinx.....	12	#warning.....	158