

PIC by example

Introduzione ai PICmicro

I **PICmicro** sono dei circuiti integrati prodotti dalla [Microchip Technology Inc.](http://www.microchip.com), che appartengono alla categoria dei microcontroller, ovvero quei componenti che integrano in un unico dispositivo tutti i circuiti necessari a realizzare un completo sistema digitale programmabile.

Come si può vedere in figura,



i PICmicro (in questo caso un PIC16F84A) si presentano esternamente come dei normali circuiti integrati TTL o CMOS, ma internamente dispongono di tutti dispositivi tipici di un sistema a microprocessore, ovvero:

- Una **CPU** (Central Processor Unit) ovvero una unità centrale di elaborazione il cui scopo è interpretare le istruzioni di programma.
- Una memoria **FLASH** in cui sono memorizzare in maniera permanente le istruzioni del programma da eseguire.
- Una memoria **RAM** (Random Access Memory) utilizzata per memorizzare le variabili utilizzate dal programma.
- Una serie di **LINEE DI I/O** (Input/Output) ovvero linee di ingresso e uscita per pilotare dispositivi esterni o ricevere impulsi da sensori, pulsanti, ecc.
- Una serie di dispositivi ausiliari al funzionamento quali generatori di clock, bus, contatori, ecc.

La presenza di tutti questi dispositivi in uno spazio estremamente contenuto, consente al progettista di avvalersi degli enormi vantaggi derivanti dall'uso di un sistema a microprocessore, anche in quei circuiti che fino a poco tempo fa erano destinati ad essere realizzati con circuiterie tradizionali.

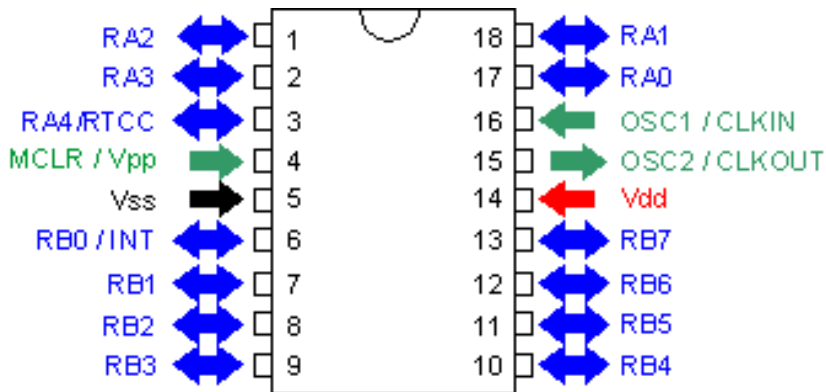
I **PICmicro** sono disponibili in un'ampia gamma di modelli per meglio adattarsi alle esigenze di progetto specifiche, differenziandosi per numero di linee di I/O e per dotazione di dispositivi. Si parte dai modelli più piccoli identificati dalla sigla **PIC12C5xx** dotati di soli 8 pin, fino ad arrivare ai modelli più grandi con sigla **PIC18Cxx** dotati di 40 e più pini.

Una descrizione dettagliata delle tipologie di PICmicro è disponibile presso il sito web della [Microchip](http://www.microchip.com) raggiungibile via internet, che consigliamo senz'altro di esplorare per la grande quantità di informazioni tecniche, software di supporto, esempi di applicazioni e aggiornamenti disponibili.

Per il nostro corso abbiamo scelto un modello intermedio di **PIC16F84A**. Esso è dotato di 18 pin di cui ben 13 disponibili per l'I/O ovvero per i collegamenti al resto del nostro circuito e di alcune caratteristiche che lo rendono maggiormente adatto alle esigenze del nostro corso.

In particolare il **PIC16F84A** dispone di una memoria per contenere il programma di tipo **FLASH** che può essere riscritta quante volte vogliamo e quindi ideale per i nostri esperimenti.

Nella figura seguente vengono riportati i pin di cui dispone il **PIC16F84A**.



Come è possibile vedere il PIC16F84A è dotato di un totale di **18 pin** disposti su due file parallele da 9 pin ciascuna. I pin contrassegnati in BLU rappresentano le linee di I/O disponibili per le nostre applicazioni, i pin in ROSSO e NERO sono i pin di alimentazione, i pin in VERDE sono riservati al funzionamento del PICmicro (MCLR per il reset e OSC1-2 per il clock).

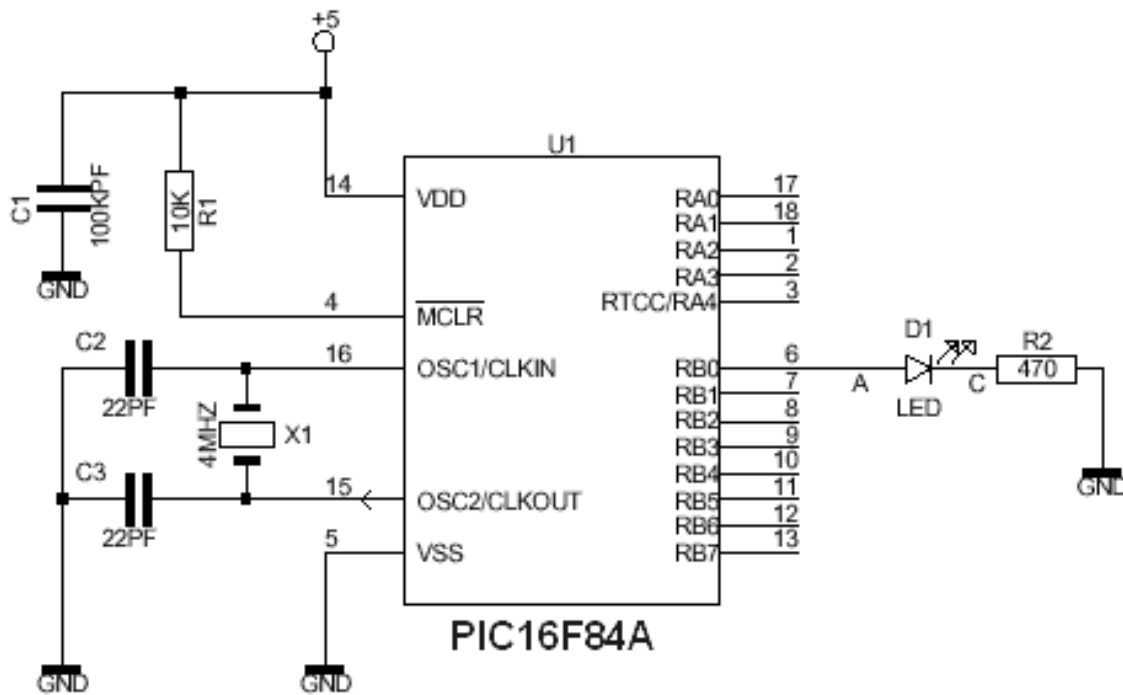
Cliccando sui singoli pin disegnati in figura è possibile visualizzare una breve descrizione del loro funzionamento.

Realizziamo un semplice lampeggiatore a led

Dopo aver visto molto brevemente cos'è e com'è fatto un PICmicro, passiamo immediatamente ad una semplice applicazione pratica.

Realizziamo un circuito molto semplice il cui scopo è quello di far lampeggiare un diodo led. Vedremo come si scrive un programma in assembler, come si compila e come si trasferisce all'interno della FLASH del PICmicro per lanciarlo in esecuzione.

Il circuito da realizzare è riportato nel seguente figura seguente:



Per alimentare il circuito è necessario fornire una tensione stabilizzata di 5 volt in continua che possiamo ottenere da un piccolo alimentatore da laboratorio oppure da un qualsiasi alimentatore autocostruito scelto tra i tanti presentati sulle varie riviste di elettronica.

La tensione di alimentazione di 5 volt viene inviata ai pin **Vdd** (pin 14) e **Vss** (pin 5) collegati rispettivamente al positivo ed alla massa.

Il pin **MCLR** (pin 4) serve per poter effettuare il reset del PICmicro. Normalmente viene mantenuto a 5 volt tramite la resistenza di pull up R1 e messo a zero quando si desidera resettare il PICmicro. Grazie alla circuiteria interna di reset di cui il PICmicro è dotato, non è necessario collegare al pin MCLR pulsanti o circuiti RC per ottenere il reset all'accensione.

I pin **OSC1/CLKIN** (pin 16) e **OSC2/CLKOUT** (pin 15) sono collegati internamente al circuito per la generazione della frequenza di clock utilizzata per temporizzare tutti i cicli di funzionamento interni al chip. Da questa frequenza dipende la quasi totalità delle operazioni interne ed in particolare la velocità con cui il PICmicro esegue le istruzioni del programma.

Nel caso del **PIC16F84A-04/P** tale frequenza può raggiungere un massimo di **4Mhz** (anche se in realtà è pratica comune spingere tale frequenza a valori molto superiori, operazione chiamata **overclocking**) da cui si ottiene una velocità di esecuzione delle istruzioni pari a **1 milione di istruzioni al secondo**. Nel nostro caso per la generazione del clock viene utilizzato un quarzo esterno da 4 MHz e due condensatori da 22pF.

Il pin **RB0** (pin 6) è una delle linee di I/O disponibili sul PICmicro per i nostri scopi. In questo caso questa linea è stata collegata ad un led tramite una resistenza di limitazione.

Scrittura e compilazione di un programma assembler

Come per qualsiasi sistema a microprocessore, anche per il PICmicro è necessario preparare un programma per farlo funzionare.

Un programma è costituito da una sequenza di istruzioni, ognuna delle quali identifica univocamente una funzione che il PICmicro deve svolgere. Ogni istruzione è rappresentata da un codice operativo (in inglese operation code o più brevemente **opcode**) composto da 14 bit ed è memorizzata in una locazione di memoria dell'area programma. Tale memoria nel PIC16F84A è di tipo FLASH e dispone di 1024 locazioni ognuna delle quali è in grado di contenere una sola istruzione oppure una coppia istruzione/dato.

Un esempio di opcode in notazione binaria viene riportato di seguito:

00 0001 0000 0000

ma è più probabile che un opcode venga rappresentato in notazione esadecimale ovvero:

0x100

che rappresenta esattamente lo stesso valore ma in forma più breve. La 0x davanti al valore indica che si tratta di una notazione esadecimale. Lo stesso valore può essere rappresentato in assembler con la notazione H'0100'.

Questi codici, completamente privi di senso per un essere umano, sono gli unici che il PICmicro è in grado di capire. Per fortuna esistono alcuni strumenti che consentono di facilitare il compito al programmatore rendendo le istruzioni più comprensibili.

Per convenzione si associa, ad ogni opcode, una breve sigla detta **mnemonica**, ovvero una sigla che aiuti a ricordare la funzione svolta da ogni istruzione.

L'opcode **0x100** dell'esempio precedente, effettua l'azzeramento del registro W (vedremo meglio di seguito che cosa significa) che in inglese viene indicato con la frase **CLEAR W REGISTER**, ovvero "AZZERA IL REGISTRO W" che nella forma abbreviata diventa **CLRW**.

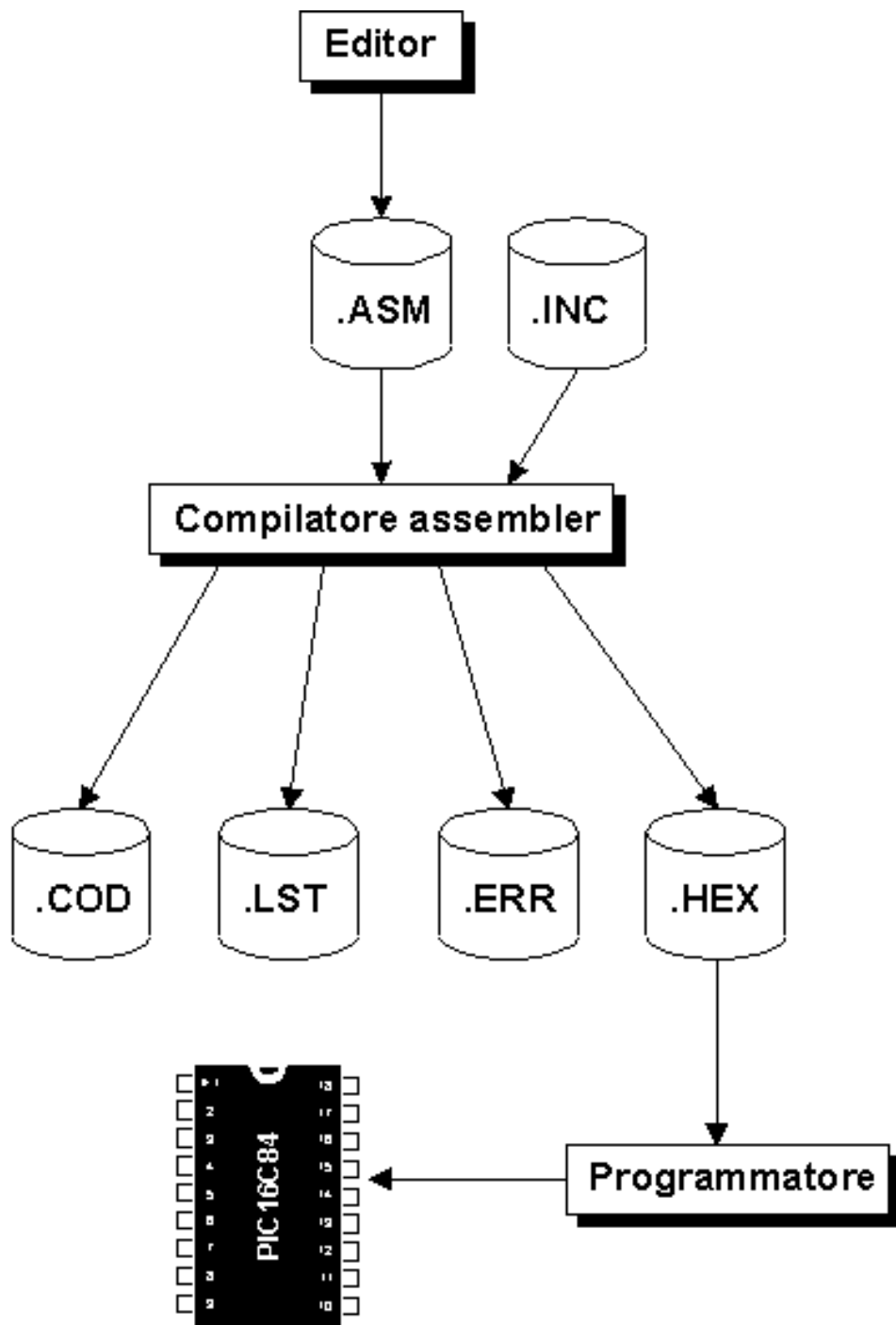
Altre sigle mnemoniche consentono di definire tutte le istruzioni che il PICmicro è in grado di eseguire ma anche variabili, costanti ed etichette (label). L'insieme di queste sigle e le regole per ordinarle per formare un programma completo viene chiamato **LINGUAGGIO ASSEMBLER**.

Per scrivere un programma in linguaggio assembler occorre conoscere le istruzioni disponibili sul micro che si intende usare (in questo caso il PICmicro), le regole sintattiche per definire variabili, parametri, ecc e disporre di un editor di testo con cui digitare il nostro programma.

Il file di testo così ottenuto viene denominato **source o sorgente assembler**.

Il passo successivo consiste nel tradurre il nostro sorgente assembler nella giusta sequenza di istruzioni in formato binario che il PICmicro è in grado di capire. Questo tipo di programma si chiama **compilatore assembler o assemblatore**.

Nella figura seguente viene schematizzato il flusso di operazioni ed i file generati necessari per ottenere un PICmicro programmato.



Come già detto, la prima operazione da effettuare è la scrittura del source assembler e la sua memorizzazione in un file di testo. L'estensione di questo file deve essere **.ASM**.

Possiamo usare allo scopo un semplice editor ASCII quale, ad esempio **NOTEPAD** di Windows anche se è vivamente consigliabile un edit ascii pensato appositamente per sviluppare sorgenti. Un ottimo programma da utilizzare sotto windows è Ultraedit32.

E' possibile generare questo file anche con programmi di elaborazione testi più sofisticati quali Word© o Star Office© avendo però l'accortezza di memorizzare sempre il file prodotto in formato testo e non in formato nativo (i .DOC tanto per intenderci). Questo per evitare che vengano memorizzati anche i caratteri di controllo della formattazione del testo che il compilatore assembler non è in grado di trattare.

Nel nostro primo esperimento pratico utilizzeremo il file [LED.ASM](#).

Il passo successivo è la compilazione del source, ovvero la trasformazione in opcode dei codici mnemonici o istruzioni assembler in esso contenute.

Il compilatore assembler che utilizzeremo è l'**MPASM.EXE** prodotto dalla Microchip.

Come è possibile vedere nella figura precedente, oltre al nostro source con estensione **.ASM** è necessario fornire al compilatore un secondo file prodotto dalla Microchip con estensione **.INC**, differente a seconda del tipo di PICmicro che stiamo utilizzando. Nel nostro caso il file è il [P16F84.INC](#). Questo source contiene alcune definizioni dipendenti dal chip utilizzato che vedremo più avanti.

Durante la compilazione, l'assemblatore genera una serie di file con il nome identico al source da cui derivano, ma con estensione diversa. Vediamo quali sono e cosa contengono:

- **.HEX** è il file contenete gli opcode da inserire nella memoria programma del PICmicro.
- **.LST** è un file di testo in cui viene riportato l'intero source assembler e la corrispondente traduzione in opcode. Non è utilizzabile per la programmazione del PICmicro ma è estremamente utile per verificare i processi di compilazione che ha eseguito l'assemblatore.
- **.ERR** contiene la lista degli errori di compilazione riscontrati ed il numero di linea all'interno del source assembler in cui sono stati rilevati.

I file **.LST** e **.ERR** vengono utilizzati per il controllo di quanto effettuato in compilazione. Solo il file **.HEX** viene utilizzato realmente per programmare il PICmicro.

Il file **.HEX** non è un file in formato binario ma un file codificato in un formato inventato dalla **Intel** per la descrizione dei file binari in formato ASCII. Senza entrare troppo nei dettagli è utile sapere che tale formato è direttamente riconoscibile da qualsiasi programmatore di PICmicro il quale provvederà a leggere da questo formato gli opcode ed a trasferirli nella memoria del PICmicro.

Analizziamo un source Assembler

Analizziamo ora linea per linea il contenuto del nostro source [LED.ASM](#). Per chi dispone di una stampante è utile effettuare una stampa del source per poter meglio seguire la nostra descrizione. Altrimenti è preferibile visualizzare il source in una finestra separata in modo da seguire simultaneamente il source e la relativa spiegazione.

Partiamo dalla prima linea di codice:

```
PROCESSOR          16F84
```

PROCESSOR è una direttiva del compilatore assembler che consente di definire per quale microprocessore è stato scritto il nostro source. Le direttive non sono delle istruzioni mnemoniche che il compilatore traduce nel rispettivo opcode, ma delle semplici indicazioni rivolte al compilatore per determinarne il funzionamento durante la compilazione. In questo caso informiamo il compilatore che le istruzioni che abbiamo inserito nel nostro source sono relative ad un PIC16F84.

```
RADIX              DEC
```

La direttiva **RADIX** serve ad informare il compilatore che i numeri riportati senza notazione, sono da intendersi come numeri decimali. Ovvero se intendiamo specificare, ad esempio il numero esadecimale 10 (16 decimale) non possiamo scrivere solamente 10 perché verrebbe interpretato come 10 decimale, ma 0x10 oppure H'10'.

ERRORLEVEL -302

La direttiva **ERRORLEVEL** serve ad escludere la segnalazione di alcuni errori di compilazione. Nel nostro caso viene utilizzata per evitare che il compilatore ci segnali questi due errori:

```
Message[302] C:\EPIC\LED.ASM 37 :
Register in operand not in bank 0.
Ensure that bank bits are correct.
```

```
Message[302] C:\EPIC\LED.ASM 40 :
Register in operand not in bank 0.
Ensure that bank bits are correct.
```

I quali ci avvertono che alla linea 37 e 40 del nostro sorgente vengono usati dei registri che non si trovano sul banco 1 e ci ricordano di commutare banco. Vedremo in seguito il significato reale di questi errori. Per ora limitiamoci a filtrarli.

INCLUDE "P16F84.INC"

Ancora un'altra direttiva per dire al compilatore la nostra intenzione di includere nel source un secondo file denominato **P16F84.INC**. Il compilatore si limiterà a sostituire la linea contenente la direttiva **INCLUDE** con il contenuto del file indicato e ad effettuare quindi la compilazione come se fosse anch'esso parte del nostro source.

```
LED EQU 0
```

Ancora direttive ! Ma quando arrivano le istruzioni ? Ancora un po di pazienza :-).

La direttiva **EQU** è molto importante in quanto ci consente di definire delle costanti simboliche all'interno del nostro source. In particolare la parola **LED** da questo punto in poi del source sarà equivalente al valore 0. Lo scopo principale dell'esistenza della direttiva **EQU** è quindi rendere i source più leggibili e consentire di cambiare i valori costanti in un unico punto del source.

E' importante notare che la parola **LED** non identifica una variabile ma semplicemente un nome simbolico valido durante la compilazione. Non sarà quindi possibile inserire istruzioni tipo **LED = 3** all'interno del source in quanto l'assegnazione dinamica di un valore ad una variabile è un'operazione che richiede l'intervento della CPU del PICmicro e che quindi deve essere espressa con istruzioni e non con direttive.

Le direttive hanno senso solo durante la compilazione del source quindi un PICmicro non potrà mai eseguire una direttiva.

Vediamo ora la linea seguente:

```
ORG 0x0C
```

Anche **ORG** è una direttiva e ci consente di definire l'indirizzo da cui vogliamo che il compilatore inizi ad allocare i dati o le istruzioni seguenti. In questo caso stiamo per definire un'area dati all'interno del PICmicro ovvero un'area in cui memorizzare variabili e contatori durante l'esecuzione del nostro programma. Quest'area coincide con l'area RAM del PICmicro definita dalla Microchip come area dei **FILE REGISTER**.

I file register altro non sono che locazioni RAM disponibili per l'utente a partire dall'indirizzo 0x0C. Questo indirizzo di inizio è fisso e non può essere cambiato in quanto le locazioni precedenti sono occupate da altri registri specializzati per uso interno.

```
Count    RES 2
```

In questa linea incontriamo una label: **Count** e una direttiva: **RES**.

La direttiva RES indica al compilatore che intendiamo riservare un certo numero di byte o meglio di file register all'interno dell'area dati; in questo caso 2 byte. La label Count, dove Count è un nome scelto da noi, è un marcatore che nel resto del source assumerà il valore dell'indirizzo in cui è stato inserito. Dato che precedentemente avevamo definito l'indirizzo di partenza a 0x0C con la direttiva ORG, Count varrà 0x0C. Se ad esempio inseriamo una label anche alla linea successiva essa varrà 0x0C + 2 (due sono i byte che abbiamo riservato) ovvero 0x0E. I nomi delle label possono essere qualsiasi ad eccezione delle parole riservate al compilatore quali sono le istruzioni mnemoniche e le direttive).

Una label si distingue da una costante simbolica perchè il suo valore viene calcolato in fase di compilazione e non assegnato da noi staticamente.

```
ORG 0x00
```

Questa seconda direttiva ORG farà riferimento ad un indirizzo in area programma (nella FLASH) anzichè in area dati. Da questo punto in poi andremo infatti ad inserire le istruzioni mnemoniche che il compilatore dovrà convertire negli opportuni opcode per il PICmicro.

Il primo opcode eseguito dal PICmicro dopo il reset è quello memorizzato nella locazione 0, da qui il valore 0x00 inserito nella ORG.

```
bsf STATUS, RP0
```

Ecco finalmente la prima istruzione mnemonica completa di parametri. I PICmicro hanno una CPU interna di tipo **RISC** per cui ogni istruzione occupa una sola locazione di memoria, opcode e parametri inclusi. In questo caso l'istruzione mnemonica **bsf** sta per **BIT SET FILE REGISTER** ovvero metti a uno (condizione logica alta) uno dei bit contenuti nella locazione di ram specificata.

Il parametro STATUS viene definito nel file [P16F84.INC](#) tramite una direttiva EQU. Il valore assegnato in questo file è 0x03 e corrisponde ad un file register (ovvero una locazione ram nell'area dati) riservato.

Anche il parametro RP0 viene definito nel file P16F84.INC con valore 0x05H e corrisponde al numero del bit che si vuole mettere a uno. Ogni file register è lungo 8 bit e la numerazione di ciascuno parte da 0 (bit meno significativo) fino ad arrivare a 7 (bit più significativo)

Questa istruzione in pratica mette a 1 il quinto bit del file register STATUS. Questa operazione è necessaria, come vedremo nelle lezioni successive, per accedere ai file register TRISA e TRISB come vedremo ora.

```
movlw B'00011111'
```

Questa istruzione sta a significare: **MOVE LITERAL TO W REGISTER** ovvero muovi un valore costante nell'accumulatore. Come avremo modo di vedere più avanti, l'accumulatore è un particolare registro utilizzato dalla CPU in tutte quelle situazioni in cui vengono effettuate operazioni tra due valori oppure in operazioni di spostamento tra locazioni di memoria. In pratica è

un registro di appoggio utilizzato dalla CPU per memorizzare temporaneamente un byte ogni volta che se ne presenta la necessità.

Il valore costante da memorizzare nell'accumulatore è **00011111** ovvero un valore binario a 8 bit dove il bit più a destra rappresenta il bit 0 o bit meno significativo.

Nell'istruzione successiva:

```
movwf TRISA
```

il valore 00011111 viene memorizzato nel registro TRISA (come per il registro STATUS anche TRISA è definito tramite una direttiva EQU) la cui funzione è quella di definire il funzionamento di ogni linea di I/O della porta A. In particolare ogni bit ad uno del registro TRISA determina un ingresso sulla rispettiva linea della porta A mentre ogni 0 determina un'uscita.

Nella seguente tabella viene riportata la configurazione che assumeranno i pin del PICmicro dopo l'esecuzione di questa istruzione:

N.bit registro TRISA	Linea porta A	N.Pin	Valore	Stato
0	RA0	17	1	Ingresso
1	RA1	18	1	Ingresso
2	RA2	1	1	Ingresso
3	RA3	2	1	Ingresso
4	RA4	3	1	Ingresso
5	-	-	0	-
6	-	-	0	-
7	-	-	0	-

Come è possibile vedere i bit 5, 6 e 7 non corrispondono a nessuna linea di I/O e quindi il loro valore non ha alcuna influenza.

Le due istruzioni successive svolgono le stesse funzioni per la porta B del PICmicro:

```
movlw B'11111110'
movwf TRISB
```

in questo caso la definizione delle linee sarà la seguente:

N.bit registro TRISB	Linea porta B	N.Pin	Valore	Stato
0	RB0	6	0	Uscita
1	RB1	7	1	Ingresso
2	RB2	8	1	Ingresso
3	RB3	9	1	Ingresso

4	RB4	10	1	Ingresso
5	RB5	11	1	Ingresso
6	RB6	12	1	Ingresso
7	RB7	13	1	Ingresso

Notate come il valore 0 nel bit 0 del registro TRISB determini la configurazione in uscita della rispettiva linea del PICmicro. Nella nostra applicazione infatti questa linea viene utilizzata per pilotare il LED da far lampeggiare.

Abbiamo visto che l'istruzione **movwf TRISB** trasferisce il valore contenuto nell'accumulatore (inizializzato opportunamente con l'istruzione **movlw B'11111110'**) nel registro TRISB. Il significato di movwf è infatti **MOVE W TO FILE REGISTER**.

bcf STATUS,RP0

Questa istruzione è simile alla **bsf** vista in precedenza, con la sola differenza che azzerà il bit anziché metterlo a uno. La sigla in questo caso è **BIT CLEAR FILE REGISTER**.

Dal punto di vista funzionale questa istruzione è stata inserita per consentire l'accesso ai registri interni del banco 0 anziché ai registri interni del banco 1 di cui fanno parte TRISA e TRISB. Una descrizione più dettagliata verrà data più avanti in questo corso.

bsf PORTB,LED

Con questa istruzione viene effettuata la prima operazione che ha qualche riscontro all'esterno del PICmicro. In particolare viene acceso il led collegato alla linea RB0. **PORTB** è una costante definita in P16F84.INC e consente di referenziare il file register corrispondente alle linee di I/O della porta B mentre **LED** è il numero della linea da mettere a 1. Se ben ricordate, all'inizio del source la costante LED è stata definita pari a 0, quindi la linea interessata sarà RB0.

MainLoop

Questa linea contiene una **label** ovvero un riferimento simbolico ad un indirizzo di memoria. Il valore della label, come detto in precedenza, viene calcolato in fase di compilazione in base al numero di istruzioni, alle direttive ORG e alle altre istruzioni che in qualche modo allocano spazio nella memoria del PICmicro. In questo caso, se contiamo le istruzioni inserite a partire dall'ultima direttiva ORG possiamo calcolare il valore che verrà assegnato a MainLoop ovvero **0x07**.

In realtà il valore che assumono le label non ha molta importanza in quanto il loro scopo è proprio quello di evitare di dover conoscere la posizione precisa degli opcode nella memoria del PICmicro permettendo comunque di referenziare una determinata locazione di memoria.

In questo caso la label MainLoop viene utilizzata come punto di ingresso di un ciclo (dall'inglese Loop) di accensione e spegnimento del led, ovvero una parte di codice che verrà ripetuta ciclicamente all'infinito. Incontreremo più avanti un riferimento a questa label.

call Delay

Questa istruzione determina una chiamata (dall'inglese *call*) ad una subroutine che inizia in corrispondenza della label **Delay**.

Le subroutine sono delle parti di programma specializzate ad effettuare una funzione specifica. Ogni qualvolta è necessaria quella funzione è sufficiente richiamarla con una sola istruzione, anziché ripetere ogni volta tutte le istruzioni necessarie ad effettuarla. In questo caso la subroutine inserisce un ritardo pari al tempo di accensione e spegnimento del led.

Le istruzioni che compongono la subroutine **Delay** sono inserite più avanti in questo stesso source.

```
btfsf PORTB, LED
```

Il significato di questa istruzione è **BIT TEST FLAG, SKIP IF CLEAR** ovvero controlla lo stato di un bit all'interno di un registro e salta l'istruzione successiva se il valore di tale bit è zero. Il bit da controllare corrisponde alla linea di uscita cui è collegato il diodo led, tramite questo test potremo determinare quindi se il led è acceso o spento e quindi agire di conseguenza, ovvero se il led è già acceso lo spegneremo, se il led è spento lo accenderemo.

```
goto SetToZero
```

Questa istruzione è un salto incondizionato (dall'inglese **GO TO**, vai a) alla label SetToZero dove troveremo le istruzioni per spegnere il led. Questa istruzione verrà saltata dall'istruzione successiva se il led è già spento.

```
bsf PORTB, LED
goto MainLoop
```

Queste due istruzioni semplicemente **accendono** il led e rimandano il programma all'ingresso del ciclo di lampeggiamento.

```
SetToZero
    bcf PORTB, LED
    goto MainLoop
```

Queste due istruzioni semplicemente **spegnono** il led e rimandano il programma all'ingresso del ciclo di lampeggiamento.

La subroutine Delay

Come descritto in precedenza questa subroutine inserisce un ritardo di circa un secondo e può essere chiamata più volte nel source tramite l'istruzione **call Delay**.

Vediamo come funziona:

```
Delay
    clrf Count
    clrf Count+1

DelayLoop
    decfsz Count, 1
    goto DelayLoop
    decfsz Count+1, 1
    goto DelayLoop
    retlw 0
```

Delay e **DelayLoop** sono due label. **Delay** identifica l'indirizzo di inizio della subroutine e viene utilizzato per le chiamate dal corpo principale del programma. **DelayLoop** viene chiamato

internamente dalla subroutine e serve come punto di ingresso per il ciclo (dall'inglese loop) di ritardo.

In pratica il ritardo viene ottenuto eseguendo migliaia di istruzioni che non fanno nulla !

Questo tipo di ritardo si chiama ritardo software o ritardo a programma. E' il tipo di ritardo più semplice da implementare e può essere utilizzato quando non è richiesto che il PICmicro esegua altri compiti mentre esegue il ritardo.

Le istruzioni:

```
clrf Count
clrf Count+1
```

CLEAR FILE REGISTER azzerano le due locazioni di ram riservate precedentemente con l'istruzione:

```
Count    RES 2
```

Queste due locazioni sono adiacenti a partire dall'indirizzo referenziato dalla label Count.

```
decfsz Count,1
```

L'istruzione significa **DECREMENT FILE REGISTER, SKIP IF ZERO** ovvero decrementa il contenuto di un registro (in questo caso Count e salta l'istruzione successiva se il valore raggiunto è zero). Se il valore raggiunto non è zero viene eseguita l'istruzione successiva:

```
goto DelayLoop
```

Che rimanda rimanda l'esecuzione all'inizio del ciclo di ritardo. Una volta raggiunto lo zero con il contatore Count vengono eseguite le istruzioni:

```
decfsz Count+1,1
goto DelayLoop
```

Che decrementano il registro seguente fino a che anche questo raggiunge lo zero. Il registro Count+1 in particolare verrà decrementato di uno ogni 256 decrementi di Count.

Quando anche Count+1 avrà raggiunto lo zero l'istruzione:

```
return
```

il cui significato è **RETURN FROM SUBROUTINE** determinerà l'uscita dalla routine di ritardo ed il proseguimento dell'esecuzione dall'istruzione successiva la call Delay.

Per finire **END** è una direttiva che indica al compilatore la fine del source assembler.

Compiliamo un source assembler

Vediamo ora come è possibile effettuare in pratica la compilazione di un source assembler.

Per prima cosa creiamo sul nostro disco fisso una directory di lavoro in cui da ora in poi

memorizzeremo tutti i file sorgenti del nostro corso. Scegliamo un nome quale ad esempio:

C:\PBE

(Qualsiasi altro nome di directory o drive è ovviamente valido. Basterà sostituire nel resto del corso tutti i riferimento a C:\PBE con il nome del drive e della directory scelti).

Copiamo ora nella nostra directory di lavoro C:\PBE il file [LED.ASM](#). Per far questo basta cliccare con il tasto destro del mouse sul nome e richiedere di salvare il file nella nostra directory di lavoro.

Scarichiamo ora da internet il compilatore MPASM disponibile gratuitamente dal sito Microchip al seguente indirizzo:

<http://www.microchip.com/1010/pline/tools/archive/other/61/index.htm>

Il file da scaricare è **asm21500.zip** (579K) e contiene l'assembler **MPASM** sia in versione Windows (MPASMWIN.EXE) che **MS/DOS** (MPASM.EXE) oltre ad una serie di utility e tutti file .INC per i vari microprocessori prodotti da Microchip .

Una volta espanso il contenuto dei file asm21500.zip nella nostra directory di lavoro possiamo lanciare il file MPASMWIN.EXE.

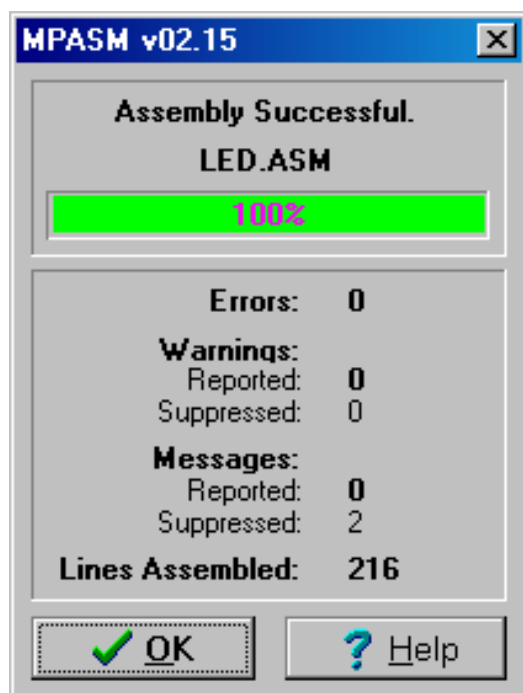
La schermata che apparirà sarà la seguente:



Premiamo il tasto "Browse..." e selezioniamo il sorgente LED.ASM. Quindi premiamo il tasto

"Assemble". L'assemblatore inizierà a compilare.

Al termine della compilazione otterremo la seguente schermata:



Se andiamo ora a vedere nella directory C:\PBE nuovi file creati, troveremo il file LED.HEX contenente il codice oggetto da utilizzare per programmare il PICmicro ed i file di supporto LED.ERR e LED.LST spiegati precedentemente.

Siamo ora pronti per programmare il PIC16F74A con il programma appena compilato.

Che programmatore usare ?

Per programmare gli esempi presentati in questo corso abbiamo scelto l'economico ma ottimo programmatore **EPIC** della **microEngineering Labs Inc.** acquistabile direttamente da questo sito [seguendo questo link](#).

Data la vastissima offerta di programmatori per PICmicro abbiamo deciso di abbandonare completamente l'ormai obsoleto **Yapp!**, presentato nelle precedenti versioni di questo corso, per presentarne uno commerciale dotato di ottime caratteristiche di velocità ed espandibilità ad un prezzo decisamente abbordabile anche per un uso hobbystico.

Qualsiasi programmatore commerciale o autocostruito in grado di programmare PIC16F84 può essere ovviamente usato con successo per seguire il corso Pic By Example. In questo caso per le modalità di programmazione dei PICmicro di prega di seguire le istruzioni di corredo al programmatore stesso.

I flag di configurazione dei PICmicro

Il PICmicro dispone di una serie di flag di configurazione contenuti nella cosiddetta **configuration word**. Questi flag determinano alcune modalità di funzionamento del PICmicro quando esso esegue un programma. La configurazione dei flag è indicata nei source d'esempio con la direttiva **__CONFIG** e dovrebbe essere letta correttamente da quasi tutti i programmatori di PICmicro. Alcuni di questi però non lo fanno per cui i flag vanno settati manualmente prima di iniziare la programmazione.

Tutti gli esercizi riportati in questo corso, salvo quando esplicitamente indicato, utilizzato la seguente configurazione:

- **Oscillatore in modalità XT.** In questa modalità il PICmicro funziona correttamente con un quarzo collegato ai pin OSC1 e OSC2 come indicato negli schemi d'esempio
- **Watch Dog Timer Disabilitato.** La funzione del Watch Dog Timer viene spiegata più avanti nel corso. Per far funzionare correttamente gli esempio (salvo quando indicato esplicitamente) il watch dog timer deve essere disabilitato.

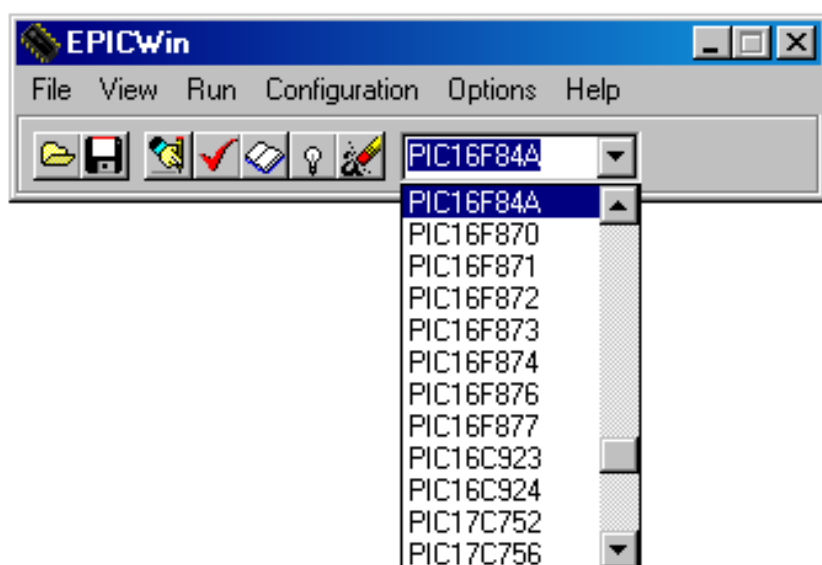
Esistono altri flag di configurazione il cui settaggio non determina cambiamenti sull'esito degli esercizi.

Programmiamo il PICmicro

Assicuriamoci di aver installato correttamente sia l'hardware che il software del nostro programmatore di PICmicro (per l'installazione hardware e software del programmatore EPIC fare riferimento a questo articolo).

Lanciamo quindi il software di gestione del programmatore (nel caso dell'EPIC il programma da lanciare in ambiente Windows è EPICWIN.EXE).

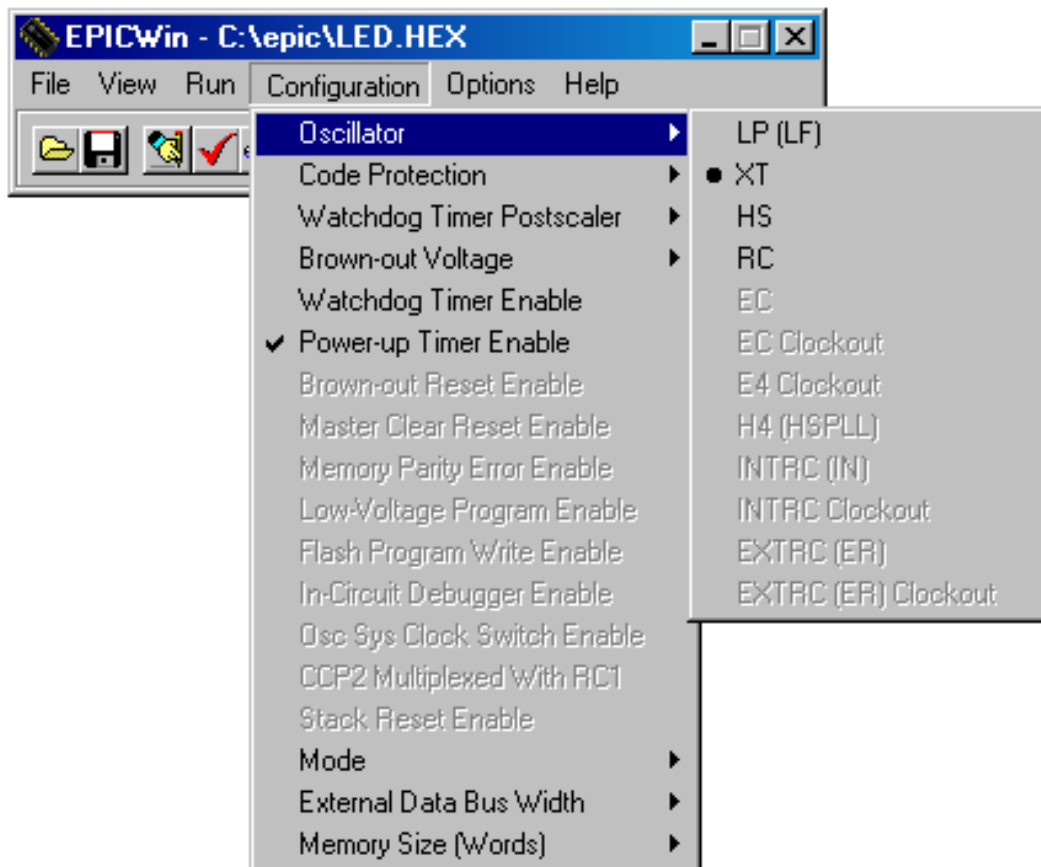
Selezioniamo il tipo di PICmicro da programmare, un PIC16F84A (in molti casi va bene anche la selezione del PIC16F84).



Inseriamo il PICmicro da programmare nella zoccolo del programmatore facendo attenzione che la disposizione del PICmicro sia corretta (Nel caso dell'EPIC bisogna far corrispondere il pin 1 del PICmicro con l'indicazione "Pin 1" riportata sulla serigrafia del circuito stampato).

Carichiamo ora il file LED.HEX dalla directory di lavoro [C:\PBE](#) (Su EPICWin si seleziona la voce "Menu File" seguita da "Open").

Assicuriamoci che il software del programmatore abbia interpretato correttamente la configurazione della configuration word.



con particolare attenzione al tipo di oscillatore che deve essere settato su XT. E sulla disabilitazione del watch dog timer. L'errata interpretazione di questi due parametri da parte del software del programmatore potrebbe portare al mancato funzionamento dei nostri esempi.

Avviamo la programmazione (Menu "Run" --> "Program"). Se non ci sono segnalazioni d'errore da parte del software del programmatore possiamo togliere il PICmicro programmato ed inserirlo nel circuito d'esempio.

Fornendo l'alimentazione dovremmo vedere il led lampeggiare.

```
;*****  
;  
; Pic by example  
; LED.ASM  
;  
;  
; (c) 2001, Sergio Tanzilli  
; http://www.tanzilli.com  
;*****  
  
        PROCESSOR      16F84  
        RADIX           DEC  
        INCLUDE         "P16F84.INC"  
        ERRORLEVEL      -302  
  
        ;Setup of PIC configuration flags  
  
        ;XT oscillator  
        ;Disable watch dog timer  
        ;Enable power up timer  
        ;Disable code protect  
  
        __CONFIG        0x3FF1  
  
LED      EQU          0  
  
        ORG             0x0C  
  
Count    RES          2  
  
        ;Reset Vector  
        ;Start point at CPU reset  
  
        ORG             0x00  
  
        bsf             STATUS,RP0  
  
        movlw           B'00011111'  
        movwf           TRISA  
  
        movlw           B'11111110'  
        movwf           TRISB  
  
        bcf             STATUS,RP0  
  
        bsf             PORTB,LED  
  
MainLoop  
  
        call            Delay  
  
        btfsc           PORTB,LED  
        goto            SetToZero  
  
        bsf             PORTB,LED  
        goto            MainLoop  
  
SetToZero  
  
        bcf             PORTB,LED  
        goto            MainLoop  
  
        ;Subroutines  
  
        ;Software delay  
  
Delay  
  
        clrf            Count  
        clrf            Count+1
```

DelayLoop

```
    decfsz    Count,1  
    goto     DelayLoop  
  
    decfsz    Count+1,1  
    goto     DelayLoop  
  
    return  
  
END
```

```
LIST
; P16F84.INC Standard Header File, Version 2.00      Microchip Technology, Inc.
NOLIST

; This header file defines configurations, registers, and other useful bits of
; information for the PIC16F84 microcontroller.  These names are taken to match
; the data sheets as closely as possible.

; Note that the processor must be selected before this file is
; included.  The processor may be selected the following ways:

;      1. Command line switch:
;          C:\ MPASM MYFILE.ASM /PIC16F84
;      2. LIST directive in the source file
;          LIST P=PIC16F84
;      3. Processor Type entry in the MPASM full-screen interface

;=====
;
;      Revision History
;
;=====

;Rev:   Date:      Reason:
;2.00   07/24/96   Renamed to reflect the name change to PIC16F84.
;1.01   05/17/96   Corrected BADRAM map
;1.00   10/31/95   Initial Release

;=====
;
;      Verify Processor
;
;=====

IFNDEF __16F84
    MESSG "Processor-header file mismatch.  Verify selected processor."
ENDIF

;=====
;
;      Register Definitions
;
;=====

W          EQU      H'0000'
F          EQU      H'0001'

;----- Register Files-----

INDF          EQU      H'0000'
TMR0          EQU      H'0001'
PCL           EQU      H'0002'
STATUS        EQU      H'0003'
FSR           EQU      H'0004'
PORTA         EQU      H'0005'
PORTB         EQU      H'0006'
EEDATA        EQU      H'0008'
EEADR         EQU      H'0009'
PCLATH        EQU      H'000A'
INTCON        EQU      H'000B'

OPTION_REG    EQU      H'0081'
TRISA         EQU      H'0085'
TRISB         EQU      H'0086'
EECON1        EQU      H'0088'
EECON2        EQU      H'0089'
```

```
;----- STATUS Bits -----
```

```
IRP          EQU      H'0007'
RP1          EQU      H'0006'
RP0          EQU      H'0005'
NOT_TO       EQU      H'0004'
NOT_PD       EQU      H'0003'
Z            EQU      H'0002'
DC           EQU      H'0001'
C            EQU      H'0000'
```

```
;----- INTCON Bits -----
```

```
GIE          EQU      H'0007'
EEIE         EQU      H'0006'
T0IE         EQU      H'0005'
INTE         EQU      H'0004'
RBIE         EQU      H'0003'
T0IF         EQU      H'0002'
INTF         EQU      H'0001'
RBIF         EQU      H'0000'
```

```
;----- OPTION Bits -----
```

```
NOT_RBPU     EQU      H'0007'
INTEDG       EQU      H'0006'
T0CS         EQU      H'0005'
T0SE         EQU      H'0004'
PSA          EQU      H'0003'
PS2          EQU      H'0002'
PS1          EQU      H'0001'
PS0          EQU      H'0000'
```

```
;----- EECON1 Bits -----
```

```
EEIF         EQU      H'0004'
WRERR        EQU      H'0003'
WREN         EQU      H'0002'
WR           EQU      H'0001'
RD           EQU      H'0000'
```

```
=====
;
;
;   RAM Definition
;
;
=====
```

```
__MAXRAM H'CF'
__BADRAM H'07', H'50'-H'7F', H'87'
```

```
=====
;
;
;   Configuration Bits
;
;
=====
```

```
_CP_ON       EQU      H'000F'
_CP_OFF      EQU      H'3FFF'
_PWRTE_ON    EQU      H'3FF7'
_PWRTE_OFF   EQU      H'3FFF'
_WDT_ON      EQU      H'3FFF'
_WDT_OFF     EQU      H'3FFB'
_LP_OSC      EQU      H'3FFC'
_XT_OSC      EQU      H'3FFD'
_HS_OSC      EQU      H'3FFE'
_RC_OSC      EQU      H'3FFF'
```

LIST

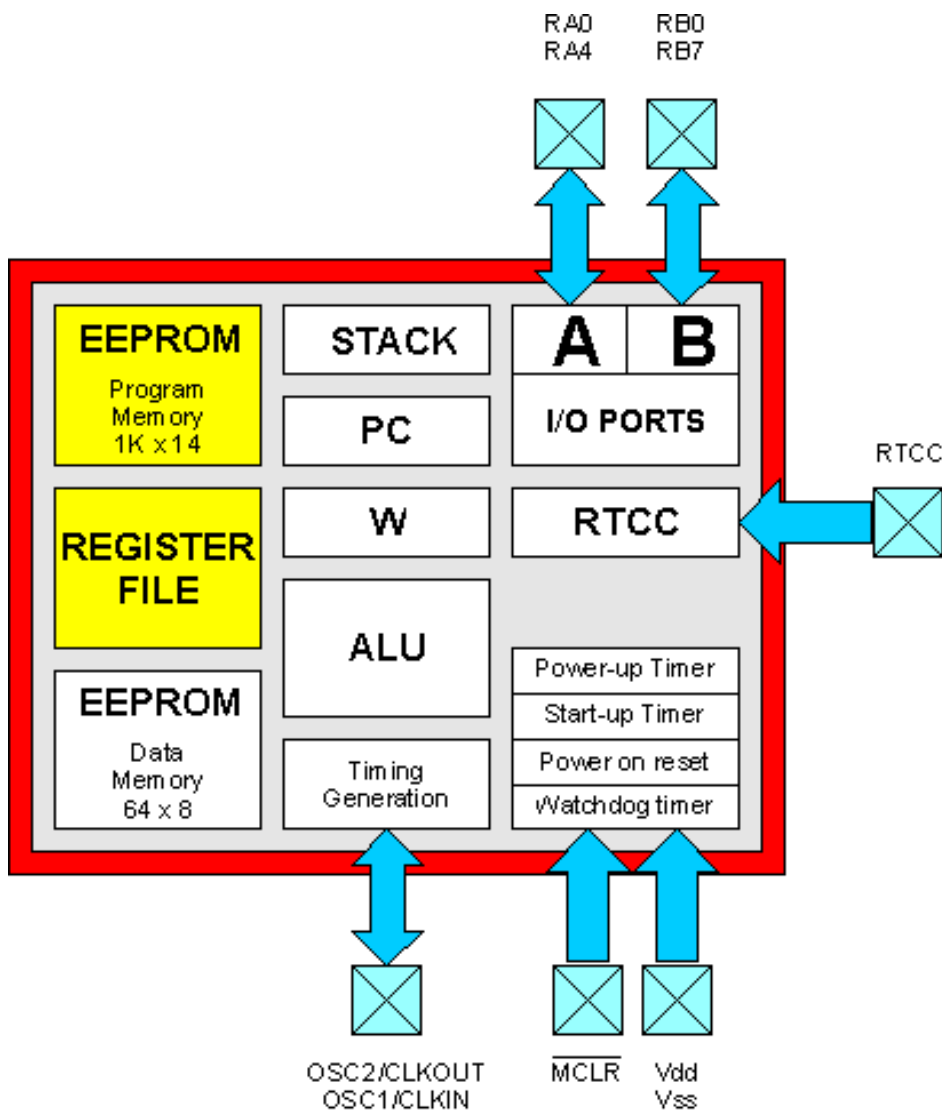
PIC by example

Architettura interna dei PIC16F84A

Dopo aver fatto un pò di pratica nella lezione precedente, passiamo ora alla teoria. Iniziamo a vedere com'è fatto internamente un PICmicro, quali dispositivi contiene e come interagiscono tra loro.

Nella figura seguente viene riprodotto lo schema a blocchi semplificato dell'architettura interna del **PIC16F84A** che ci aiuterà a seguire meglio quanto verrà spiegato di seguito. Le parti evidenziate in giallo, sono le componenti che di volta in volta andremo ad analizzare.

Iniziamo dalla memoria programma o **PROGRAM MEMORY** e dalla memoria dati o **REGISTER FILE**.



La Program Memory

La PROGRAM MEMORY è una memoria speciale di tipo FLASH, non cancellabile elettricamente, ed utilizzata nel PICmicro per tenere memorizzato il programma da eseguire.

La sua capacità di memorizzazione è di **1024 locazioni** ognuna in grado di contenere un **opcode a 14 bit** ovvero una istruzione base del PICmicro. Il programma più complesso che potremo realizzare non potrà essere quindi più lungo di 1024 istruzioni.

Gli indirizzi riservati alla PROGRAM MEMORY vanno da **0x000** a **0x3FF** (0x3FF esadecimale = 1023

decimale).

Il PICmicro può solamente eseguire le istruzioni memorizzate in queste locazioni. Non può in alcun modo leggere, scrivere o cancellare quanto in esse contenuto. Questo vale in particolar modo per i PIC16F84A mentre su altri modelli quali i **PIC16F87x** è possibile anche aggiornare la memoria programma a run time ovvero a programma in esecuzione.

Per scrivere, leggere e cancellare queste locazioni è necessario un dispositivo esterno denominato **programmatore**. Un esempio di programmatore è l'[EPIC Plus](#) il o **PICSTART** prodotto dalla Microchip o molti altri ancora disponibili in commercio.

La prima locazione di memoria, all'indirizzo zero, deve contenere la prima istruzione che il PICmicro dovrà eseguire al reset e per questo viene nominata **Reset Vector**.

Come ricorderete, nel source [LED.ASM](#) presentato nella lezione precedente era stata inserita la direttiva:

```
ORG 0x00
```

per segnare l'inizio del programma. Questa direttiva tiene conto del fatto che l'esecuzione del programma al reset parte dall'indirizzo 0x000 dell'area programma.

L'istruzione che segue immediatamente la direttiva ORG 0x00:

```
bsf STATUS, RP0
```

sarà quindi la prima istruzione ad essere eseguita.

Il Register File

Il **REGISTER FILE** è un'insieme di locazioni di memoria **RAM** ovvero memorie con cui è possibile leggere e modificare il contenuto senza l'ausilio di programmatori esterni e direttamente dal programma in esecuzione sul PICmicro.

Date le sue caratteristiche il REGISTER FILE è la memoria normalmente utilizzata per memorizzare le variabili di programma, ovvero tutti quei valori il cui contenuto varia durante l'esecuzione.

Contrariamente alla PROGRAM MEMORY il REGISTER FILE perde il suo contenuto quando il PICmicro viene spento per cui è necessario reinizializzare i valori di tutte le sue locazioni prima di poterla usare.

Alcune locazioni del REGISTER FILE ed in particolare quelle che si trovano agli indirizzi più bassi vengono usate per il controllo dell'hardware del PICmicro secondo quanto illustrato di seguito.

Le locazioni di memoria presenti nel REGISTER FILE sono indirizzabili direttamente in uno spazio di memoria che va da **0x00** a **0x2F** per un totale di **48 byte**, denominato **pagina 0**. Un secondo spazio di indirizzamento denominato **pagina 1** va da **0x80** a **0xAF**. Per accedere a questo secondo spazio è necessario ricorrere ai due bit ausiliari **RP0** e **RP1** secondo le modalità che andremo a spiegare più avanti.

Le prime **12 locazioni** della pagina 0 (da **0x00** a **0x0B**) e della pagina 1 (da **0x80** a **0x8B**) sono quelle riservate alle funzioni speciali per il funzionamento del PICmicro e non possono essere utilizzate per altri scopi.

Se, ad esempio, volessimo definire tutte le linee della porta B del PIC in uscita agendo sul registro **TRISB**, potremmo scegliere di referenziare direttamente il registro con il suo indirizzo:

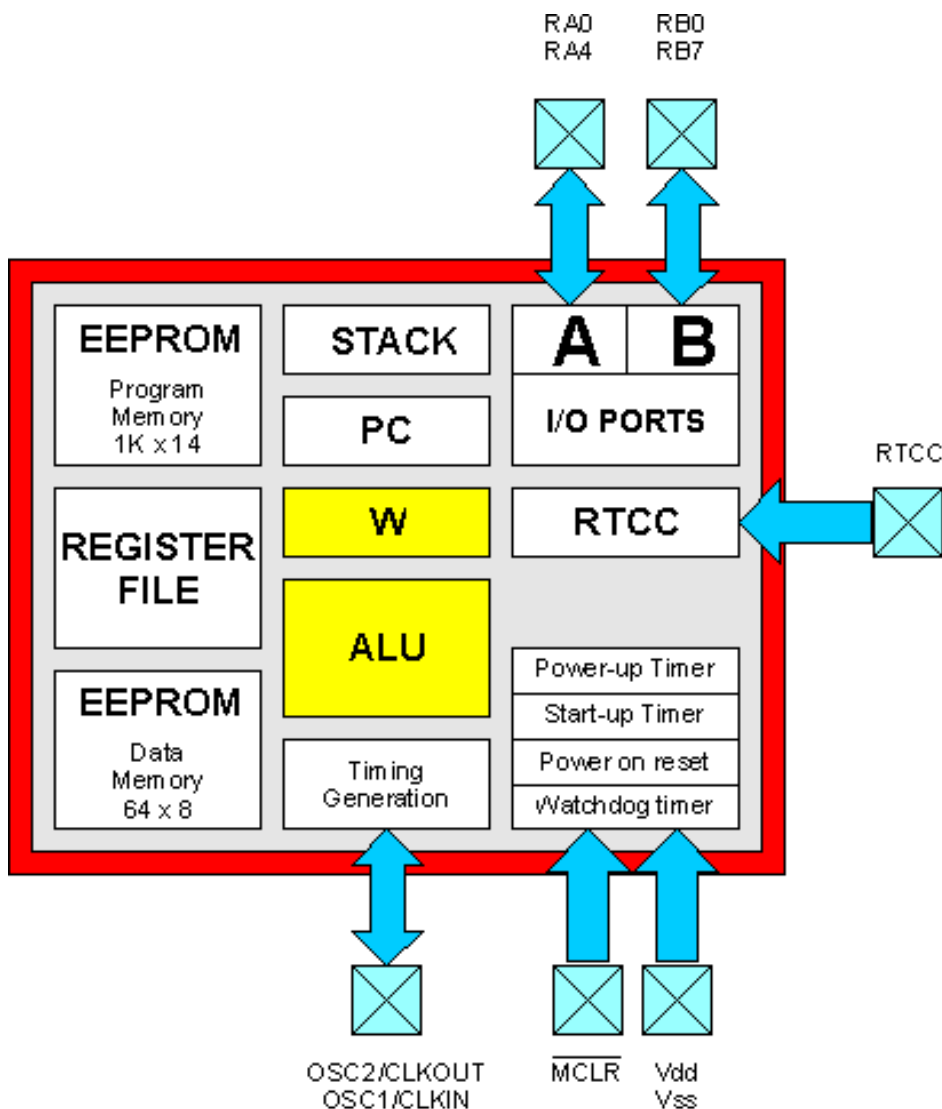
```
movlw B'00000000'
movwf 0x86
```

oppure, referenziare lo stesso registro con il suo nome simbolico:

```
movlw B'00000000'
movwf TRISB
```

avendo però l'accortezza di inserire la direttiva **INCLUDE "P16F84.INC"** nel nostro source.

La ALU



La **ALU** (acronimo di **A**rithmetic and **L**ogic **U**nit ovvero unità aritmetica e logica) è la componente più complessa del PICmicro in quanto contiene tutta la circuiteria delegata a svolgere le funzioni di calcolo e manipolazione dei dati durante l'esecuzione di un programma.

La **ALU** è una componente presente in tutti i microprocessori e da essa dipende direttamente la potenza di calcolo del micro stesso.

La **ALU** del **PIC16F84A** è in grado di operare su valori ad **8 bit**, ovvero valori numerici non più grandi di 255. Esistono microprocessori con ALU a 16, 32, 64 bit e oltre. La famiglia Intel® 80386®, 486® e Pentium® ad esempio dispone di una ALU a 32 bit. Le potenze di calcolo raggiunte da questi micro sono

notevolmente superiori a scapito della complessità della circuiteria interna ed accessoria e conseguentemente dello spazio occupato.

L'Accumulatore o registro W

Direttamente connesso con la ALU c'è il **registro W** denominato anche **accumulatore**. Questo registro consiste di una semplice locazione di memoria in grado di contenere un solo valore a 8 bit.

La differenza sostanziale tra il registro W e le altre locazioni di memoria consiste proprio nel fatto che, per referenziare il registro W, la ALU non deve fornire nessun indirizzo di memoria, ma può accedere direttamente.

Il registro W viene utilizzato molto spesso nei programmi per PICmicro.

Facciamo un esempio pratico. Supponiamo di voler inserire nella locazione di memoria **0xC** del **REGISTER FILE** il valore **0x01**. Cercando tra le istruzioni del PICmicro ci accorgiamo subito che non esiste un'unica istruzione in grado di effettuare questa operazione ma dobbiamo necessariamente ricorrere all'accumulatore ed usare due istruzioni in sequenza.

Vediamo perché:

Come detto nei passi precedenti, l'opcode di una istruzione non può essere più grande di **14 bit** mentre a noi ne servirebbero:

8 bit per specificare il valore che intendiamo inserire nella locazione di memoria,
7 bit per specificare in quale locazione di memoria vogliamo inserire il nostro valore,
6 bit per specificare quale istruzione intendiamo utilizzare.

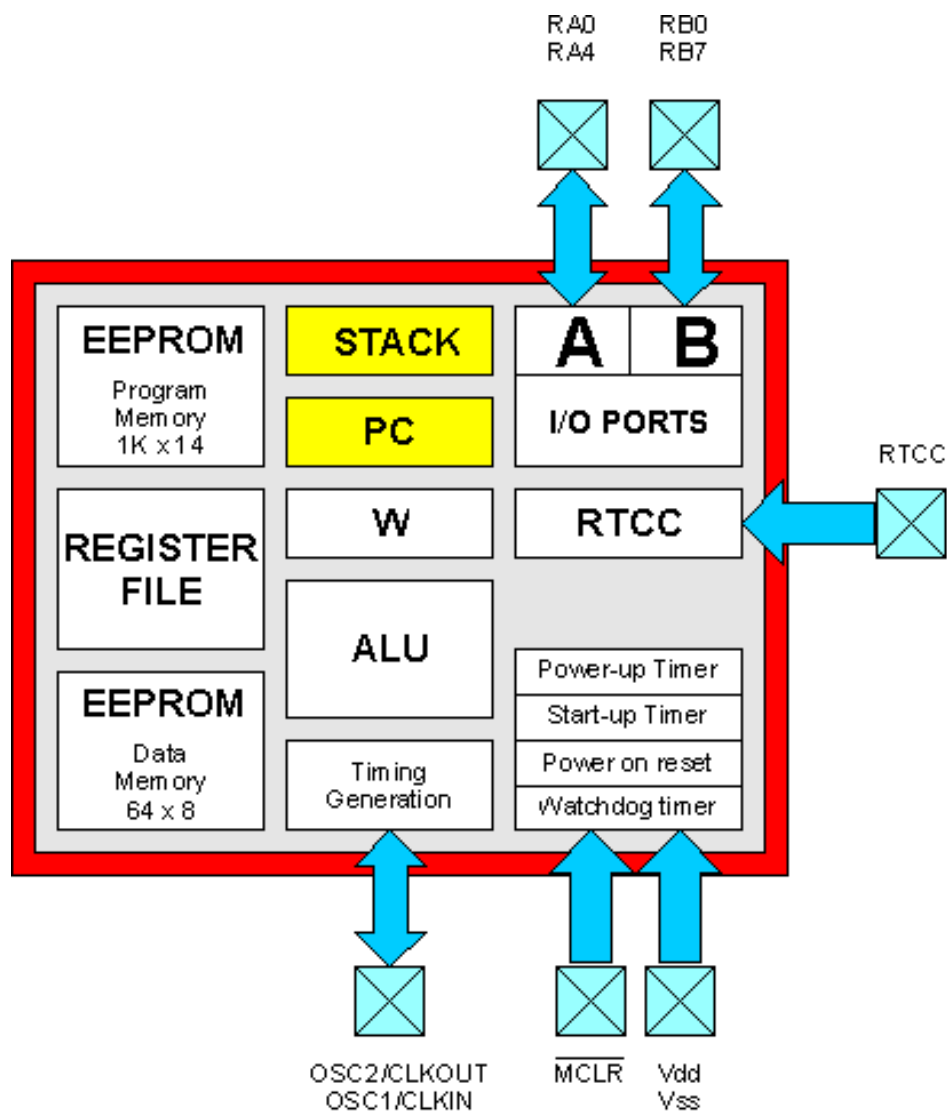
per un totale di **8 + 7 + 6 = 21 bit**.

Dobbiamo quindi ricorrere a due istruzioni, ovvero:

```
movlw    0x01
movwf    0x0C
```

che prima inseriscono nel **registro W** il valore **0x01H** con l'istruzione **MOVe Literal to W** e poi lo "muovono" nella locazione 0x0C con l'istruzione **MOVe W to F**.

Il Program Counter (PC)



Come abbiamo già visto nei passi precedenti, il PIC16F84A inizia l'esecuzione del programma a partire dal vettore di reset (**Reset Vector**) ovvero dall'istruzione memorizzata nella prima locazione di memoria (indirizzo 0x000).

Dopo aver eseguito questa prima istruzione passa quindi all'istruzione successiva memorizzata nella locazione 0x001 e così via. Se non esistesse nessuna istruzione in grado di influenzare in qualche modo l'esecuzione del programma, il PICmicro arriverebbe presto ad eseguire tutte le istruzioni presenti nella sua memoria fino all'ultima locazione disponibile.

Sappiamo ovviamente che non è così e che qualsiasi microprocessore o linguaggio di programmazione dispone di istruzioni di salto, ovvero di istruzioni in grado di modificare il flusso di esecuzione del programma in base alle esigenze del programmatore.

Una di queste istruzioni è la **GOTO** (dall'inglese **GO TO**, vai a) che ci permette di cambiare la sequenza di esecuzione e di "saltare" direttamente ad un qualsiasi punto, all'interno della memoria programma, e di continuare quindi l'esecuzione a partire da quel punto.

Facciamo un esempio:

```

ORG      0x00

Point1
    movlw 10
    goto  Point1
  
```

Al reset il PICmicro eseguirà l'istruzione **MOVLW 10** memorizzata alla locazione 0x000, la quale inserirà nell'accumulatore il valore decimale 10, quindi passerà ad eseguire l'istruzione successiva **GOTO Point1**.

Questa istruzione determinerà un salto incondizionato alla locazione di memoria puntata dalla label Point1 ovvero di nuovo alla locazione 0x000. Nel suo insieme quindi, questo programma non farà altro che eseguire continuamente le due istruzioni elencate.

Durante questo ciclo (o loop), per determinare quale sarà l'istruzione successiva da eseguire, il PIC utilizza uno speciale registro denominato **PROGRAM COUNTER** (dall'inglese contatore di programma) la cui funzione è proprio quella di mantenere traccia dell'indirizzo che contiene la prossima istruzione da eseguire.

Questo registro viene incrementato automaticamente ad ogni istruzione eseguita per determinare il passaggio all'istruzione successiva. Al momento del RESET del PIC il PROGRAM COUNTER viene azzerato, determinando così l'inizio dell'esecuzione a partire dall'indirizzo 0x000.

L'istruzione GOTO consente l'inserimento a programma di un nuovo valore nel PROGRAM COUNTER ed il di conseguente salto ad una locazione qualsiasi dell'area programma del PIC.

Lo Stack Pointer

Un'altra istruzione molto utile, che influenza il valore del PROGRAM COUNTER è la **CALL** (dall'inglese chiamata) con la quale è possibile effettuare delle CHIAMATE A SUBROUTINE.

Questa istruzione funziona in maniera molto simile alla GOTO. Come la GOTO infatti permette di scrivere nel PROGRAM COUNTER un nuovo indirizzo di esecuzione del programma. La differenza sostanziale consiste però nel fatto che prima di eseguire il salto, il PIC memorizza, in un altro registro speciale, denominato **STACK**, l'indirizzo di quella che sarebbe dovuta essere la successiva istruzione da eseguire se non si fosse incontrata la CALL.

Vediamo meglio con un esempio:

```

ORG      0x00

Point1
    movlw    10
    call     Point2
    goto     Point1

Point2
    movlw    11
    return
  
```

In questo caso il PICmicro, dopo aver eseguito la MOVLW 10 passa ad eseguire l'istruzione CALL Point2. Prima di saltare però, memorizza nello STACK l'indirizzo 0x002, ovvero l'indirizzo della locazione successiva alla CALL. L'esecuzione passa quindi all'istruzione MOVLW 11 e quindi alla istruzione RETURN (dall'inglese ritorno). Questa istruzione, come dice il suo nome, consente di "ritornare", ovvero di riprendere l'esecuzione a partire dall'istruzione successiva alla CALL che aveva determinato l'abbandono del flusso principale del programma utilizzando il valore memorizzato nel registro di STACK.

Come detto l'operazione appena effettuata viene denominata CHIAMATA A SUBROUTINE, ovvero una interruzione momentanea del normale flusso di programma per "chiamare" in esecuzione una serie di istruzioni per poi ritornare al normale flusso di esecuzione.

La parola STACK in inglese significa "**catasta**" ed infatti su questa catasta è possibile depositare, uno sull'altro, più indirizzi per recuperarli quando servono. Questo tipo di memorizzazione viene anche denominata **LIFO** dall'inglese **Last In First Out**, in cui l'ultimo elemento inserito (last in) deve necessariamente essere il primo ad uscire (last out). Grazie a questa caratteristica è possibile effettuare più CALL annidate ovvero l'una nell'altra e mantenere sempre traccia del punto in cui riprendere il flusso al momento che si incontra una istruzione RETURN.

Vediamo un altro esempio:

```

ORG      0x00

Point1
    movlw    10
    call     Point2
    goto     Point1

Point2
    movlw    11
    call     Point3
    return

Point3
    movlw    12
    return

```

In questo caso nella subroutine Point2 viene effettuata un'ulteriore CALL alla subroutine Point3. Al ritorno da quest'ultima il programma dovrà rientrare nella subroutine Point2 eseguire la RETURN e quindi tornare nel flusso principale.

Gli indirizzi da memorizzare nello stack sono due in quanto viene incontrata una seconda CALL prima ancora di incontrare la RETURN corrispondente alla prima.

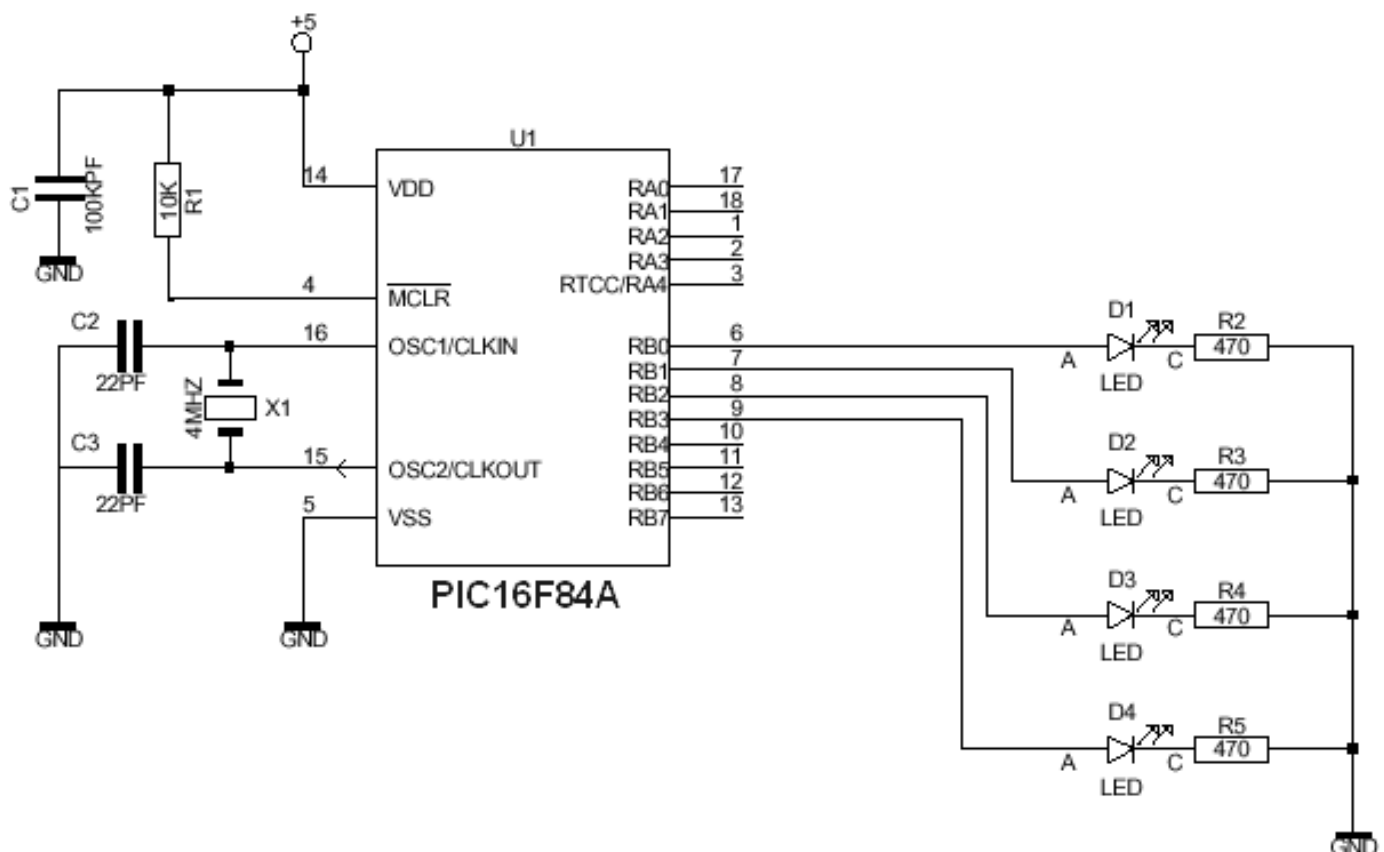
Il PIC16F84A dispone di uno stack a 8 livelli, ovvero uno stack che consente fino ad 8 chiamate annidate..

E' importante assicurarsi, durante la stesura di un programma, che ci sia sempre una istruzione RETURN per ogni CALL per evitare pericolosi disallineamenti dello stack che in esecuzione possono dar adito a errori difficilmente rilevabili.

Realizziamo le "luci in sequenza"

Proviamo ora a fissare i concetti finora appresi rielaborando il source [LED.ASM](#) presentato nella prima lezione per realizzare un lampeggiatore sequenziale a quattro led. Il nuovo source modificato si chiamerà [SEQ.ASM](#).

Nella figura seguente viene riportato lo schema elettrico del nuovo circuito, sostanzialmente equivalente al circuito presentato nella prima lezione, con l'unica variante che ora i led collegati sono quattro anziché uno.



Le linee di I/O utilizzate sono **RB0** per primo led, **RB1** per il secondo, **RB2** per il terzo ed **RB3** per il quarto. Esse vanno quindi configurate tutte in uscita all'inizio del programma cambiando le istruzioni:

```
movlw B'11111110'
movwf TRISB
```

in

```
movlw B'11110000'
movwf TRISB
```

in cui i quattro bit meno significativi, corrispondenti alle linee RB0,1,2,3 vengono messi a zero per definire tali linee in uscita.

Nell'area di memoria del REGISTER FILE (che nel source inizia con la direttiva **ORG 0x0C**) oltre ai due byte referenziati dalla label Count, riserviamo un ulteriore byte con label Shift che utilizzeremo per determinare la sequenza di accensione dei led. La direttiva da inserire è:

```
Shift RES 1
```

Prima di eseguire il ciclo principale (label MainLoop) iniziamo il nuovo registro Shift a 00000001B con le seguenti istruzioni:

```
movlw B'00000001'
movwf Shift
```

A questo punto, nel ciclo principale del nostro programma, ci occuperemo di trasferire il valore memorizzato nel registro Shift sulla Porta B ottenendo quindi l'accensione del primo led, con le seguenti istruzioni:

```

movf    Shift,W
movwf   PORTB

```

quindi di effettuare lo shift a sinistra del valore contenuto in Shift di un bit, con le seguenti istruzioni:

```

bcf     STATUS,C
rlf     Shift,F

```

la prima istruzione serve ad azzerare il bit CARRY del registro di stato STATUS che verrà analizzato nelle lezioni successive. L'istruzione RLF **R**otate **L**eft **F** through Carry (ruota a sinistra attraverso il bit di carry) sposta di un bit verso sinistra il valore memorizzato nel registro Shift inserendo nella posizione occupata dal bit 0 il valore del bit di Carry (che come già detto andremo a vedere in seguito). Per far sì che il bit inserito sia sempre zero viene eseguita prima della RLF l'istruzione BCF STATUS,C per azzerare questo bit.

A questo punto il registro Shift varrà 00000010, quindi, al ciclo successivo, una volta trasferito tale valore sulla port B si otterrà lo spegnimento del LED1 e l'accensione del LED2 e così via per i cicli successivi.

Quando il bit 4 di Shift varrà 1, vorrà dire che tutti e quattro i led sono stati accesi almeno una volta e occorre quindi ripartire dal led 1. Le istruzioni seguenti svolgono questo tipo di controllo:

```

btfsc   Shift,4
swapf   Shift,F

```

L'istruzione BTFSC Shift,4 controlla appunto se il bit 4 del registro Shift vale 1. Se si esegue l'istruzione successiva SWAPF Shift,F altrimenti la salta.

L'istruzione SWAP (dall'inglese "scambia") in pratica scambia i quattro bit più significativi contenuti nel registro Shift con i quattro meno significativi. Dal valore iniziale del registro Shift pari a 00010000 ottenuto dopo alcune ripetizioni del ciclo MainLoop si ottiene il valore 00000001 ed in pratica alla riaccensione del primo led.

```
;
;*****
; Pic by example
; SEQ.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****
```

```
PROCESSOR      16F84
RADIX          DEC
INCLUDE        "P16F84.INC"
ERRORLEVEL    -302
```

```
;Setup of PIC configuration flags
```

```
;XT oscillator
;Disable watch dog timer
;Enable power up timer
;Disable code protect
```

```
__CONFIG      0x3FF1
```

```
ORG          0x0C
```

```
Count      RES      2
Shift      RES      1
```

```
;Reset Vector
;Program start point at CPU reset
```

```
ORG          0x00
```

```
bsf          STATUS,RP0
```

```
movlw       B'00011111'
movwf       TRISA
```

```
movlw       B'11110000'
movwf       TRISB
```

```
bcf          STATUS,RP0
```

```
movlw       B'00000001'
movwf       Shift
```

MainLoop

```
movf        Shift,W
movwf       PORTB
```

```
bcf          STATUS,C
rlf          Shift,F
```

```
btfsc       Shift,4
swapf       Shift,F
```

```
call        Delay
```

```
goto        MainLoop
```

```
; Subroutines
```

Delay

```
clrf        Count
clrf        Count+1
```

DelayLoop

```
decfsz      Count,1
```

```
goto    DelayLoop

decfsz  Count+1,1
goto    DelayLoop

return

END
```

PIC by example

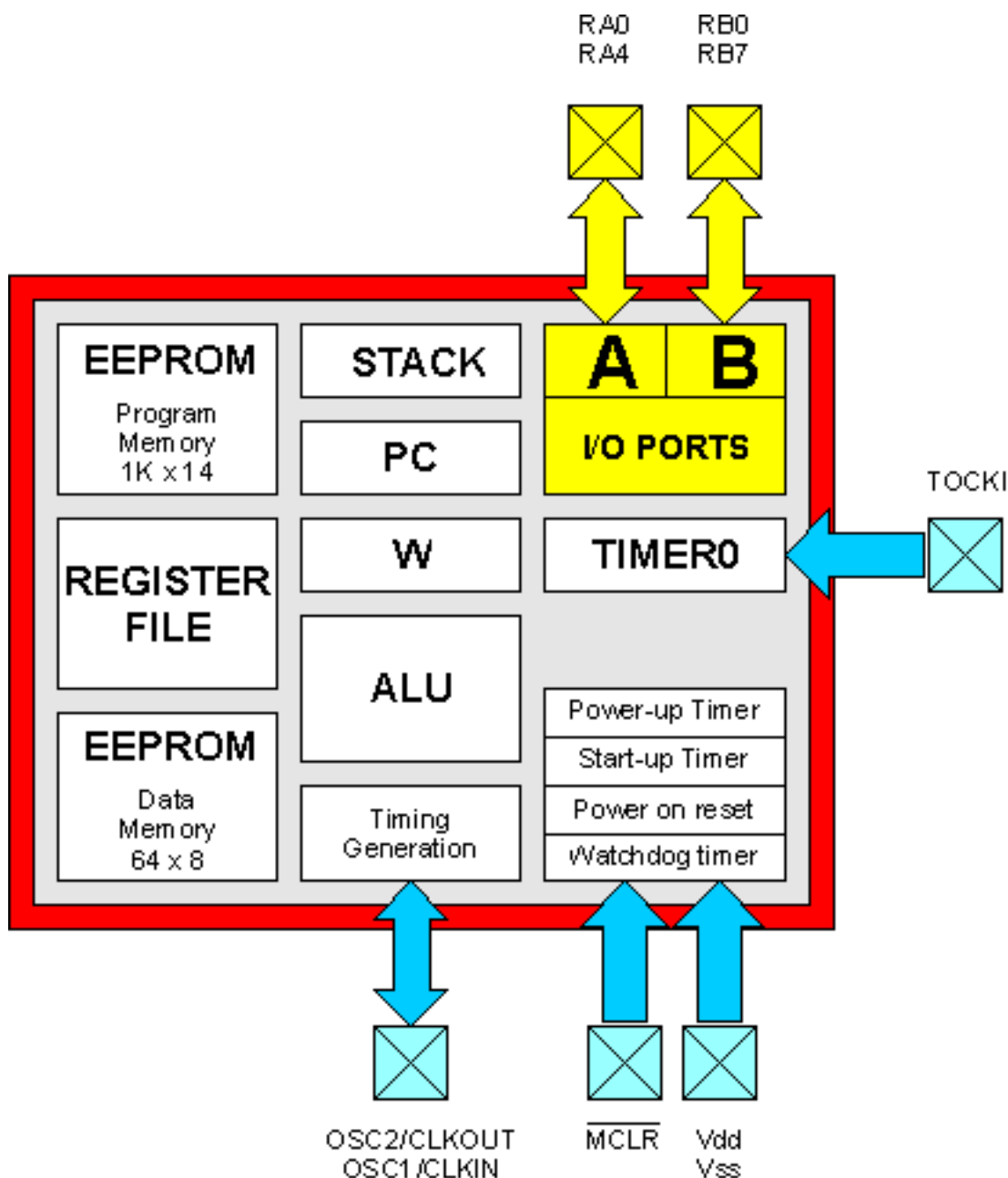
Introduzione alle periferiche

Le porte A e B

Il PIC16F84A dispone di un totale di 13 linee di I/O organizzate in due porte denominate **PORTA A** e **PORTA B**.

La PORTA A dispone di 5 linee configurabili sia in ingresso che in uscita identificate dalle sigle RA0, RA1, RA2, RA3 ed RA4.

La PORTA B dispone di 8 linee anch'esse configurabili sia in ingresso che in uscita identificate dalle sigle RB0, RB1, RB2, RB3, RB4, RB5, RB6 ed RB7.



La suddivisione delle linee in due porte distinte è dettata dai vincoli dell'architettura interna del PIC16F84A che prevede la gestione di dati di lunghezza massima pari a 8 bit.

Per la gestione delle linee di I/O da programma, il PIC dispone di due registri interni per ogni porta denominati **TRISA** e **PORTA** per la porta A e **TRISB** e **PORTB** per la porta B.

I registri TRIS A e B, determinano il funzionamento in ingresso o in uscita di ogni singola linea, i registri PORT A e B determinano lo stato delle linee in uscita o riportano lo stato delle linee in ingresso.

Ognuno dei bit contenuti nei registri menzionati corrisponde univocamente ad una linea di I/O.

Ad esempio il bit 0 del registro PORTA e del registro TRIS A corrispondono alla linea RA0 , il bit 1 alla linea RA1 e così via.

Se il bit 0 del registro TRISA viene messo a zero, la linea RA0 verrà configurata come linea in uscita, quindi il valore a cui verrà messo il bit 0 del registro PORTA determinerà lo stato logico di tale linea (0 = 0 volt, 1 = 5 volt).

Se il bit 0 del registro TRISA viene messo a uno, la linea RA0 verrà configurata come linea in ingresso, quindi lo stato logico in cui verrà posta dalla circuiteria esterna la linea RA0 si rifletterà sullo stato del bit 0 del registro PORTA.

Facciamo un esempio pratico, ipotizziamo di voler collegare un led sulla linea RB0 ed uno switch sulla linea RB4, il codice da scrivere sarà il seguente:

```
movlw    00010000B
tris     B
```

in cui viene messo a 0 il bit 0 (linea RB0 in uscita) e a 1 il bit 4 (linea RB4) in ingresso. Si ricorda a tale proposito che nella notazione binaria dell'assembler il bit più a destra corrisponde con il bit meno significativo quindi il bit 0.

Per accendere il led dovremo scrivere il seguente codice:

```
bsf      PORTB, 0
```

Per spegnerlo:

```
bcf      PORTB, 0
```

Per leggere lo stato dello switch collegato alla linea RB4, il codice sarà:

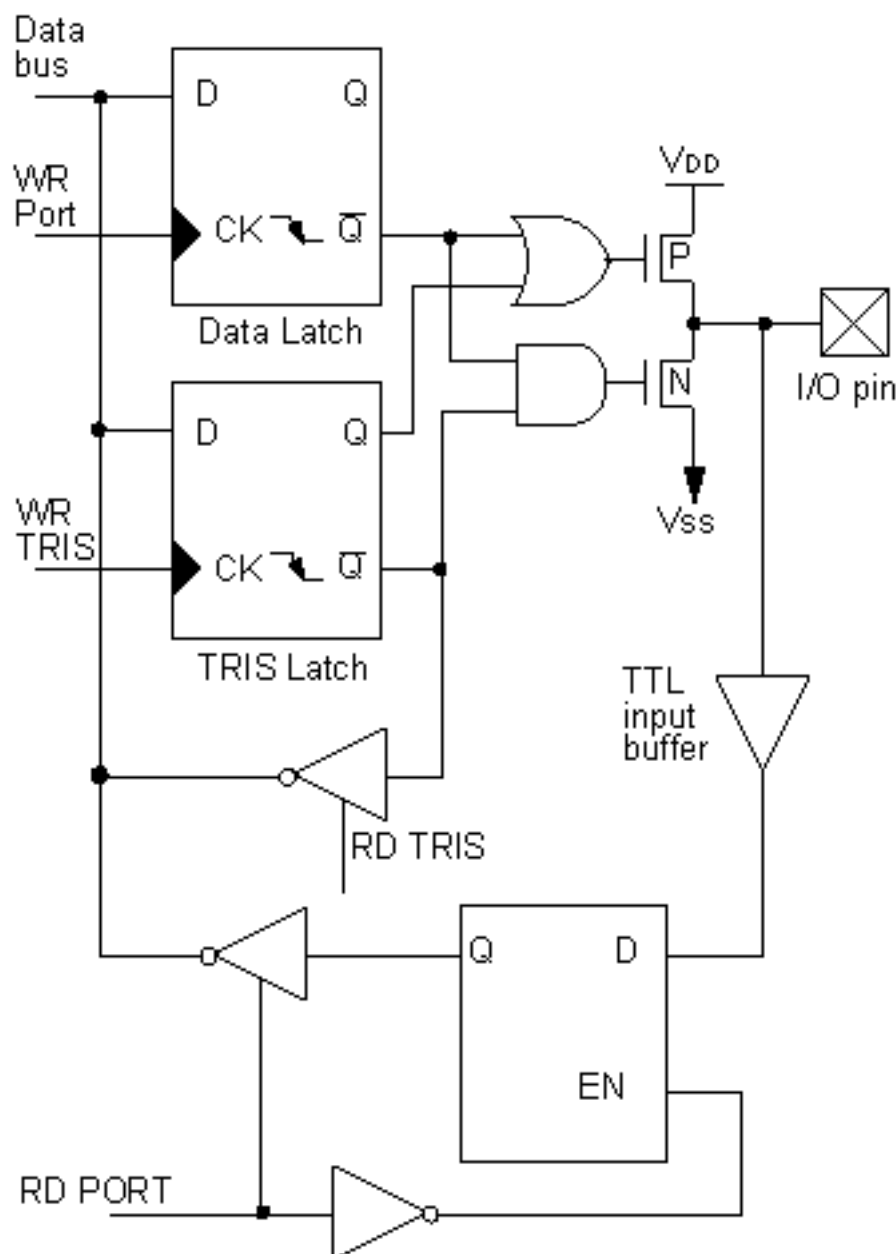
```
btfss    PORTB, 4
goto     SwitchAMassa
goto     SwitchAlPositivo
```

Stadi d'uscita delle linee di I/O

Per rendere più adattabili i PICmicro alle diverse esigenze di utilizzo, la Microchip ha implementato diverse tipologie di stati d'uscita per le linee di I/O. Esistono quindi dei gruppi di pin il cui comportamento è leggermente differenziato da altri gruppi. Conoscendo meglio il funzionamento dei diversi stadi d'uscita potremo sfruttare al meglio le loro caratteristiche ed ottimizzare il loro uso nei nostri progetti.

Stadio d'uscita delle linee RA0, RA1, RA2 e RA3

Iniziamo dal gruppo di linee RA0, RA1, RA2 ed RA3 per le quali riproduciamo, nella figura seguente, lo schema dello stadio d'uscita estratto dal data sheet della Microchip:



Come accennato al passo precedente, la configurazione di una linea come ingresso o uscita dipende dallo stato dei bit nel registro TRIS (TRISA per la porta A e TRISB per la porta B).

Prendiamo come esempio la linea RA0 ed analizziamo il funzionamento dello stadio

d'uscita sia quando la linea funziona in ingresso, che quando funziona in uscita.

Funzionamento in ingresso

Per configurare la linea RA0 in ingresso, dobbiamo mettere a 1 il bit 0 del registro TRISA con l'istruzione:

```
bsf      TRISA, 0
```

Questo determina una commutazione ad 1 dello stato logico del flip-flop di tipo D-latch indicato nel blocco con il nome TRIS latch. Per ogni linea di I/O esiste uno di questi flip-flop e lo stato logico in cui si trova dipende strettamente dallo stato logico del relativo bit nel registro TRIS (anzi per meglio dire ogni bit del registro TRIS è fisicamente implementato con un TRIS latch).

L'uscita Q del TRIS latch è collegata all'ingresso di una porta logica di tipo OR. Questo significa che, indipendentemente dal valore presente all'altro ingresso, l'uscita della porta OR varrà sempre 1 in quanto uno dei suoi ingressi vale 1 (vedi tavola della verità). In questa condizione il transistor P non conduce e mantiene la linea RA0 scollegata dal positivo d'alimentazione.

Allo stesso modo l'uscita negata del TRIS latch è collegata all'ingresso di una porta AND quindi l'uscita di questa varrà sempre 0 in quanto uno dei suoi ingressi vale 0 (vedi tavola). In questa condizione anche il transistor N non conduce mantenendo la linea RA0 scollegata anche dalla massa. Lo stato logico della linea RA0 dipenderà esclusivamente dalla circuiteria esterna a cui la collegheremo.

Applicando 0 o 5 volt al pin RA0, sarà possibile leggerne lo stato sfruttando la circuiteria d'ingresso del blocco rappresentata dal TTL input buffer e dal latch d'ingresso.

Funzionamento in uscita

Per configurare la linea RA0 in uscita, dobbiamo mettere a 0 il bit 0 del registro TRISA con l'istruzione:

```
bcf      TRISA, 0
```

Questo determina la commutazione a 0 dell'uscita Q del TRIS latch (ed a 1 dell'uscita Q negata). In questo stato il valore in uscita dalle porte OR e AND dipende esclusivamente dallo stato dell'uscita Q negata del Data Latch. Come per il TRIS latch, anche il Data Latch dipende dallo stato di un bit in un registro, in particolare del registro PORTA. La sua uscita negata viene inviata all'ingresso delle due porte logiche OR e AND e quindi direttamente sulla base dei transistor P ed N.

Se mettiamo a 0 il bit 0 del registro PORTA con l'istruzione:

```
bcf      PORTA, 0
```

otterremo la conduzione del transistor N con conseguente messa a 0 della linea RA0. Se

invece mettiamo a 1 il bit 0 con l'istruzione:

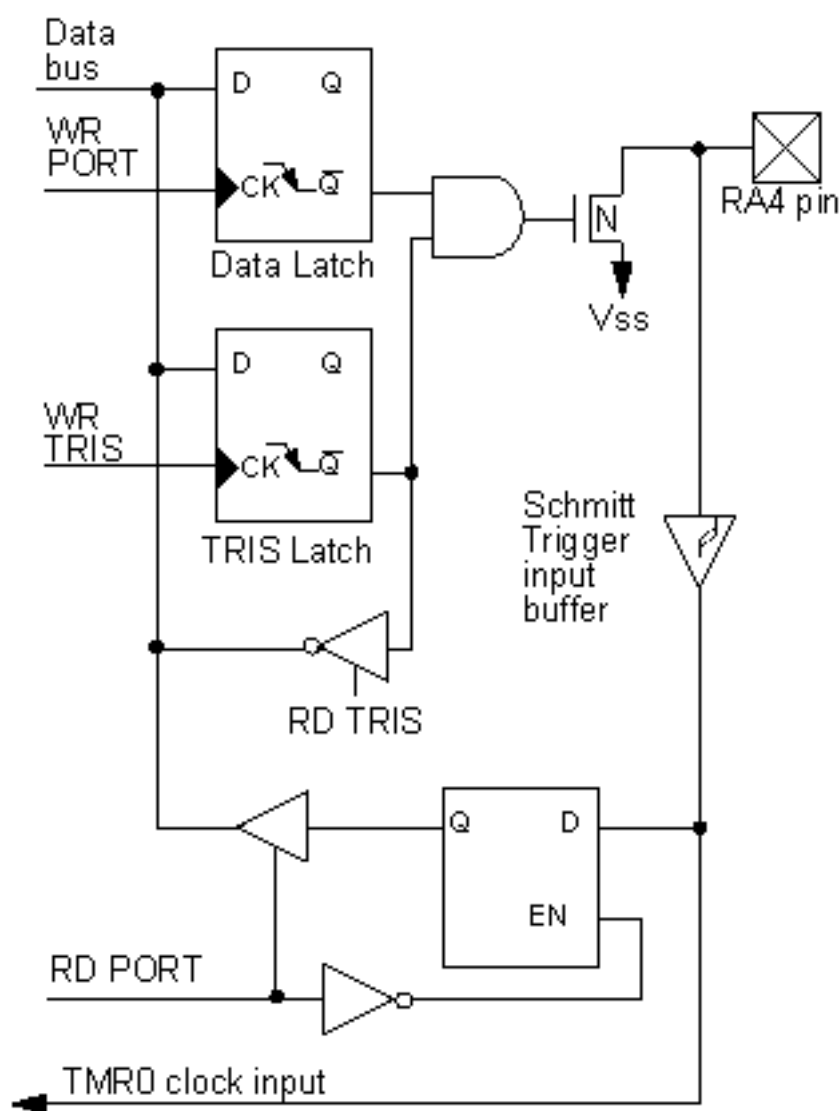
```
bsf    PORTA, 0
```

otterremo la conduzione del transistor P con conseguenza messa a +5 volt della linea RA0. In questa condizione è sempre possibile rileggere il valore inviato sulla linea tramite la circuiteria d'ingresso.

Stadio d'uscita della linea RA4

Analizziamo ora il funzionamento dello stadio d'uscita della linea RA4 che si differenzia da tutte le altre linee di I/O in quanto condivide lo stesso pin del PIC16F84 con il TOCKI.

Nella figura seguente viene riprodotto lo schema a blocchi dello stadio d'uscita estratto dal data sheet Microchip:



La logica di commutazione è sostanzialmente identica al gruppo di linee RA0-3 ad eccezione dell'assenza della porta OR e del transistor P, ovvero di tutta la catena che consente di collegare al positivo la linea RA4. Questo significa, in termini pratici, che quando la linea RA4 viene programmata in uscita e messa a 1 in realtà non viene

connessa al positivo ma rimane scollegata. Tale tipo di circuiteria d'uscita viene denominata a "collettore aperto" ed è utile per applicazioni in cui sia necessario condividere uno stesso collegamento con più pin d'uscita e ci sia quindi la necessità di mettere in alta impedenza una linea d'uscita senza doverla riprogrammare come linea d'ingresso.

Se vogliamo essere sicuri che la linea RA4 vada a 1 dovremo collegare esternamente una resistenza di pull-up, ovvero una resistenza collegata al positivo di alimentazione.

Vedremo in seguito l'utilizzo della linea indicata sullo schema TMR0 clock input.

Stadio d'uscita delle linee RB0, RB1, RB2 ed RB3

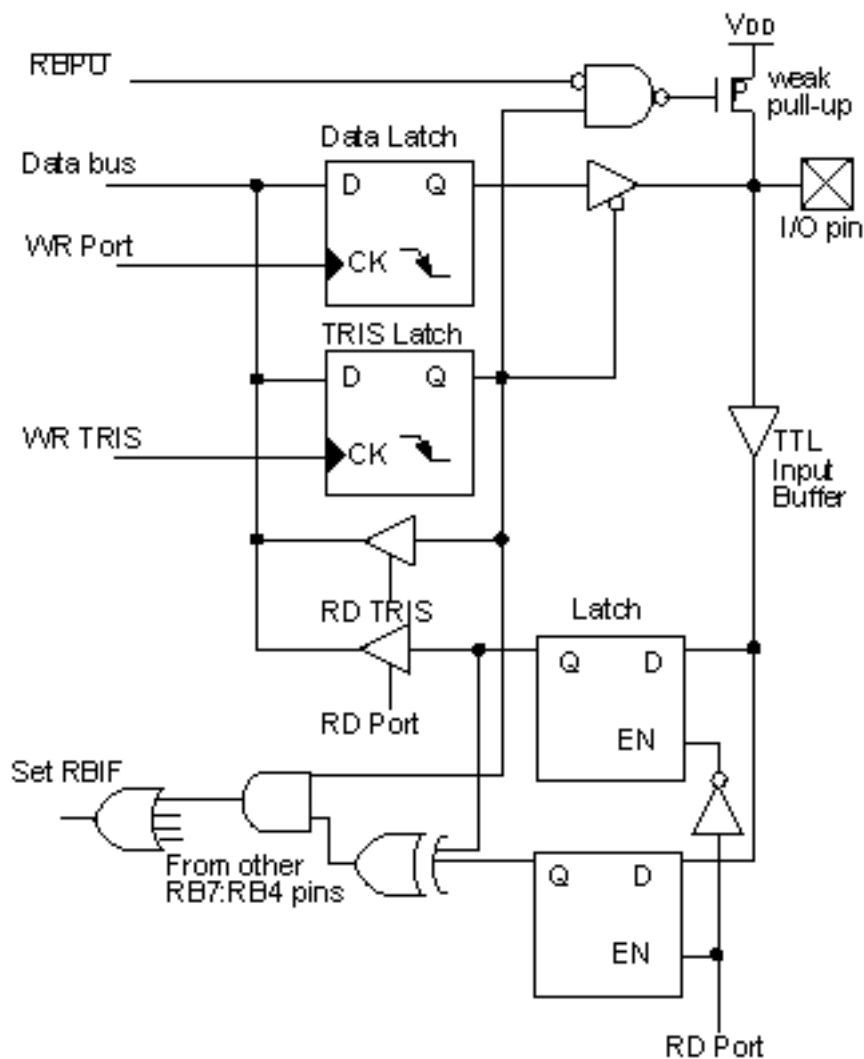
Anche per questo gruppo di linee rimane sostanzialmente invariata la logica di commutazione. Esse dispongono in aggiunta una circuiteria di weak pull-up attivabile quando le linee sono programmate in ingresso.

In ingresso infatti, come spiegato precedentemente, le linee vengono completamente scollegate dal PIC in quanto sia il transistor P che il transistor N sono aperti. Lo stato delle linee dipende quindi esclusivamente dalla circuiteria esterna. Se tale circuiteria è di tipo a collettore aperto o più semplicemente è costituita da un semplice pulsante che, quando premuto, collega a massa la linea di I/O, è necessario inserire una resistenza di pull-up verso il positivo per essere sicuri che quando il pulsante è rilasciato ci sia una condizione logica a 1 stabile sulla linea d'ingresso. La circuiteria di weak pull-up consente di evitare l'uso di resistenze di pull-up e può essere attivata o disattivata agendo sul bit RBPU del registro OPTION .

Nella figura seguente viene riprodotto lo schema a blocchi dello stadio d'uscita estratto dal data sheet Microchip:

Stadio d'uscita delle linee RB4, RB5, RB6 e RB7

Nella figura seguente viene riprodotto lo schema a blocchi dello stadio d'uscita estratto dal data sheet Microchip:



Input da tastiera

Dopo aver realizzato, nella lezione precedente, le luci in sequenza sfruttando le linee da RB0 a RB3 come linee di output, vediamo ora come si può realizzare un input da tastiera configurando le linee da RB4 a RB7 come linee di input.

Per far questo ampliamo il circuito presentato nella lezione 2 con quattro pulsanti da stampato denominati SW1, SW2, SW3 ed SW4 e collegati secondo lo schema elettrico riportato di seguito:

[Schema elettrico](#)

Ognuno di questi pulsanti collega a massa una linea di ingresso normalmente mantenuta a + 5 volt da una resistenza (da R6 a R9). Prendendo, ad esempio, il pin 10 del PIC16F84A, questa linea verrà mantenuta a +5 volt finché non verrà premuto il tasto SW1 che provvederà a portare la linea ad 0 volt.

Realizziamo un programma d'esempio che accenda ciascuno dei led D1, D2, D3 e D4 in corrispondenza della pressione di uno dei tasti SW1, SW2, SW3 e SW4.

Il source dell'esempio è riportato nel file [INPUT.ASM](#).

La parte iniziale del programma esegue le stesse funzioni effettuate negli esempi precedenti ed in particolare le istruzioni:

```
movlw 11110000B
movwf TRISB
```

configurano le linee da RB0 a RB3 in uscita per il collegamento con i led e le linee da RB4 a RB7 in ingresso per il collegamento con i quattro pulsanti. Il resto del programma.

L'istruzione:

```
bcf STATUS,RP0
```

Effettua uno swap sul banco di registri 0 in modo che possiamo accedere direttamente allo stato delle linee di I/O.

MainLoop

```
clrf PORTB
```

Questa istruzione spegne tutti i led collegati sulla PORTA B and ogni ciclo di loop in modo che possano poi essere accesi sulla base dello stato dei pulsanti.

```
btfss PORTB,SW1
bsf PORTB,LED1
```

Queste due istruzioni vengono eseguite per ogni linea collegata ad un pulsante per verificare se il pulsante è premuto e per accendere il led corrispondente.

in pratica la:

```
btfss PORTB,SW1
```

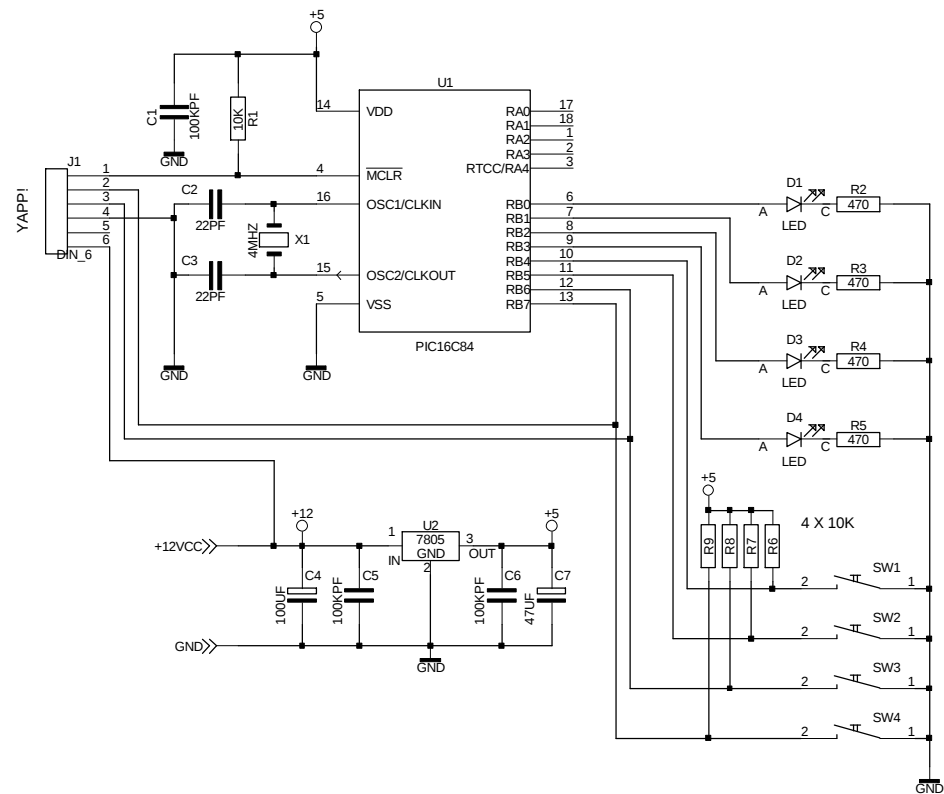
salta la successiva:

```
bsf PORTB,LED1
```

solo se il pulsante SW1 è rilasciato. In caso contrario la esegue accendendo il led. Questa coppia di istruzioni viene eseguita per ogni tasto.

Il tutto viene eseguito all'interno di un singolo loop tramite l'istruzione:

```
goto MainLoop
```

WWW.PICPOINT.COM		
PROJECT: PIC BY EXAMPLE		
NOTE: EXAMPLE N.3		
ENGINEER: SERGIO TANZILLI		
VER: 1.0	DATE: 2 NOV 1998	PAGES: 1 OF: 1

```
;*****
;
; Pic by example
; INPUT.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****
```

```
PROCESSOR      16F84
RADIX          DEC
INCLUDE        "P16F84.INC"
```

```
;Setup of PIC configuration flags
```

```
;XT oscillator
;Disable watch dog timer
;Enable power up timer
;Disable code protect
```

```
__CONFIG      3FF1H
```

```
LED1 EQU 0
LED2 EQU 1
LED3 EQU 2
LED4 EQU 3
SW1 EQU 4
SW2 EQU 5
SW3 EQU 6
SW4 EQU 7
```

```
ORG 0CH
```

```
;Reset Vector
;Punto di inizio del programma al reset della CPU
```

```
ORG 00H
```

```
;Commuta sul secondo banco dei registri per accedere ai registri TRISA e TRISB
```

```
bsf STATUS,RP0
```

```
;Definizione delle linee di I/O (0=Uscita, 1=Ingresso)
```

```
;Definizione della porta A
```

```
movlw 00011111B
movwf TRISA & 7FH
```

```
;Definizione della porta B
```

```
;Le linee da RB0 a RB3 vengono programmate in uscita per essere collegate ai quattro
```

led

```
;Le linee da RB4 a RB7 vengono programmate in ingresso per essere collegate ai quattro
```

pulsanti

```
movlw 11110000B
movwf TRISB & 7FH
```

```
;Commuta sul primo banco dei registri
```

```
bcf STATUS,RP0
```

MainLoop

```
;Spegne tutti i led
clrf PORTB
```

```
;Se e' premuto il pulsante SW1 accende il LED1
```

```
btfss PORTB,SW1
bsf PORTB,LED1
```

```
;Se e' premuto il pulsante SW2 accende il LED2
btfss    PORTB,SW2
bsf      PORTB,LED2

;Se e' premuto il pulsante SW3 accende il LED3
btfss    PORTB,SW3
bsf      PORTB,LED3

;Se e' premuto il pulsante SW4 accende il LED4
btfss    PORTB,SW4
bsf      PORTB,LED4

goto     MainLoop

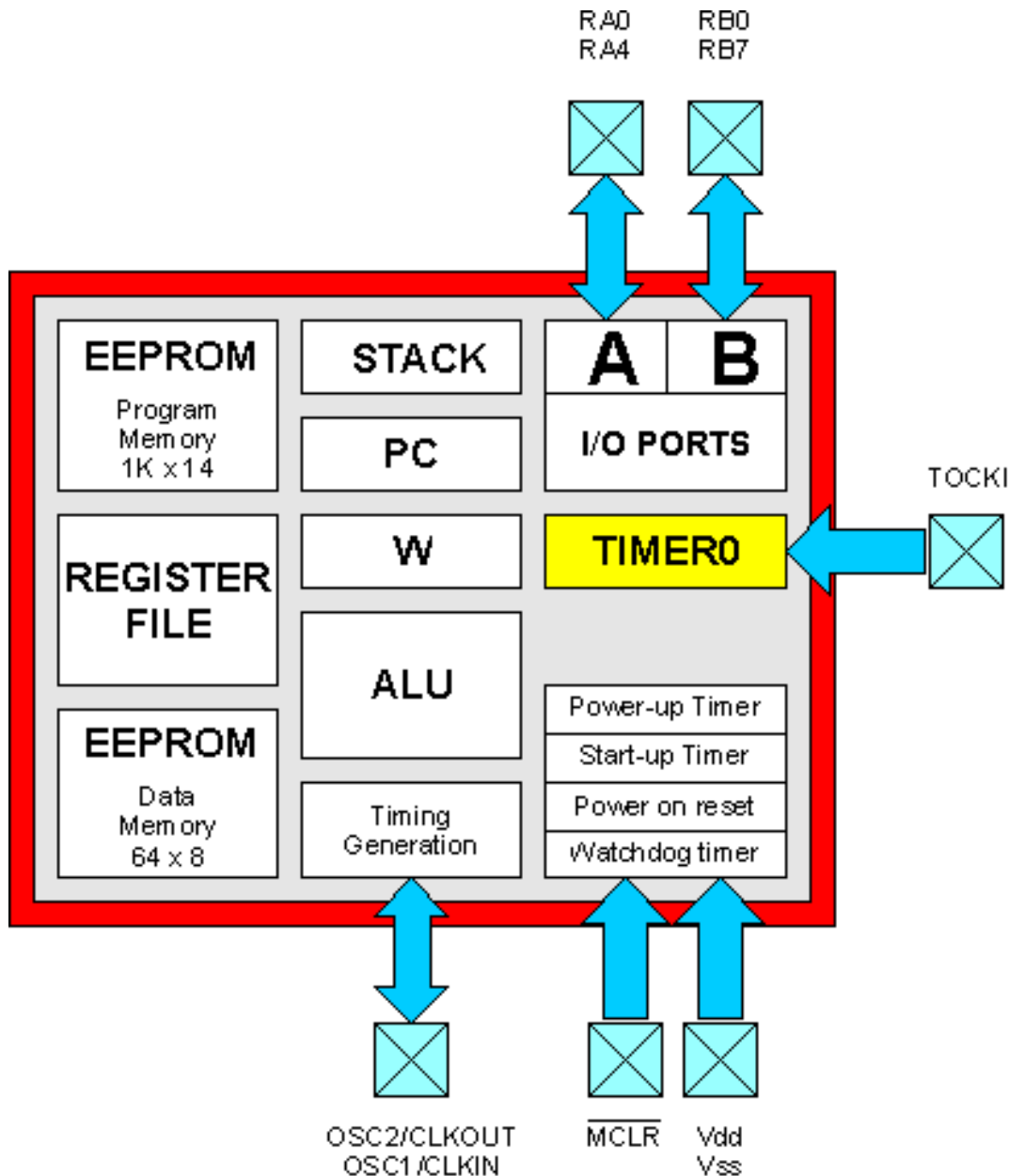
END
```

PIC by example

Il contatore TMR0 ed il PRESCALER

Il registro contatore TMR0

Vediamo ora cosa è e come funziona il registro TMR0.



Il registro TMR0 è un contatore, ovvero un particolare tipo di registro il cui contenuto viene incrementato con cadenza regolare e programmabile direttamente dall'hardware del PIC. In pratica, a differenza di altri registri, il TMR0 non mantiene inalterato il valore che gli viene memorizzato, ma lo incrementa continuamente, se ad esempio scriviamo in esso il valore 10 con le seguenti istruzioni:

```
movlw    10
movwf    TMR0
```

dopo un tempo pari a quattro cicli macchina, il contenuto del registro comincia ad essere incrementato a 11, 12, 13 e così via con cadenza costante e del tutto indipendente dall'esecuzione del resto del programma.

Se, ad esempio, dopo aver inserito un valore nel registro TMR0, eseguiamo un loop infinito

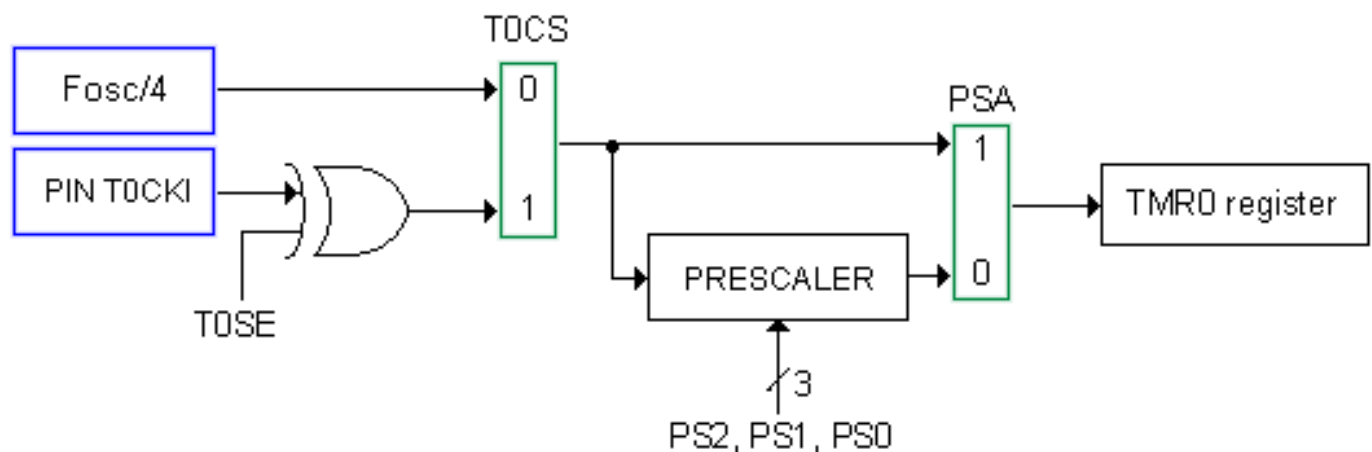
```
movlw    10
movwf    TMR0
loop
    goto loop
```

il registro TMR0 viene comunque incrementato dall'hardware interno al PIC contemporaneamente all'esecuzione del loop.

Una volta raggiunto il valore 255 il registro TMR0 viene azzerato automaticamente riprendendo quindi il conteggio non dal valore originariamente impostato ma da zero.

La frequenza di conteggio è direttamente proporzionale alla frequenza di clock applicata al chip e può essere modificata programmando opportunamente alcuni bit di configurazione.

Nella figura seguente viene riportata la catena di blocchi interni al PIC che determinano il funzionamento del registro TMR0.



I blocchi Fosc/4 e T0CKI riportati in blu rappresentano le due possibili sorgenti di segnale per il contatore TMR0.

Fosc/4 è un segnale generato internamente al PIC dal circuito di clock ed è pari alla frequenza di clock divisa per quattro.

T0CKI è un segnale generato da un eventuale circuito esterno ed applicato al pin T0CKI corrispondente al pin 3 nel PIC16F84A.

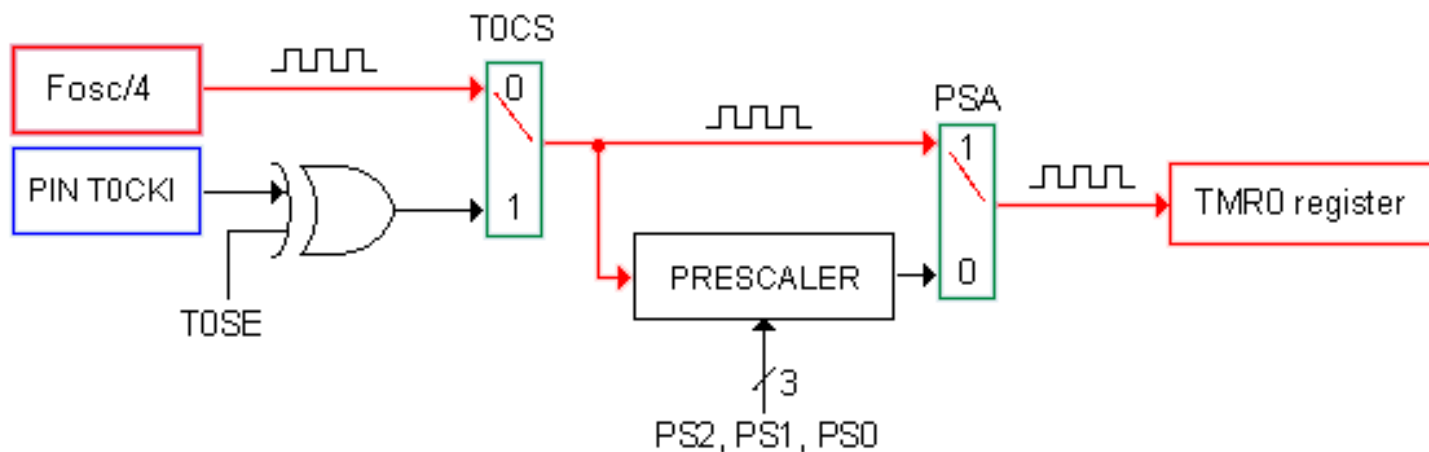
I blocchi T0CS e PSA riportati in verde sono due commutatori di segnale sulla cui uscita viene presentato uno dei due segnali in ingresso in base al valore dei bit T0CS e PSA del registro OPTION.

Il blocco PRESCALER è un divisore programmabile il cui funzionamento verrà spiegato nel

prossimo passo.

Vediamo in pratica come è possibile agire su questi blocchi per ottenere differenti modalità di conteggio per il registro TMR0.

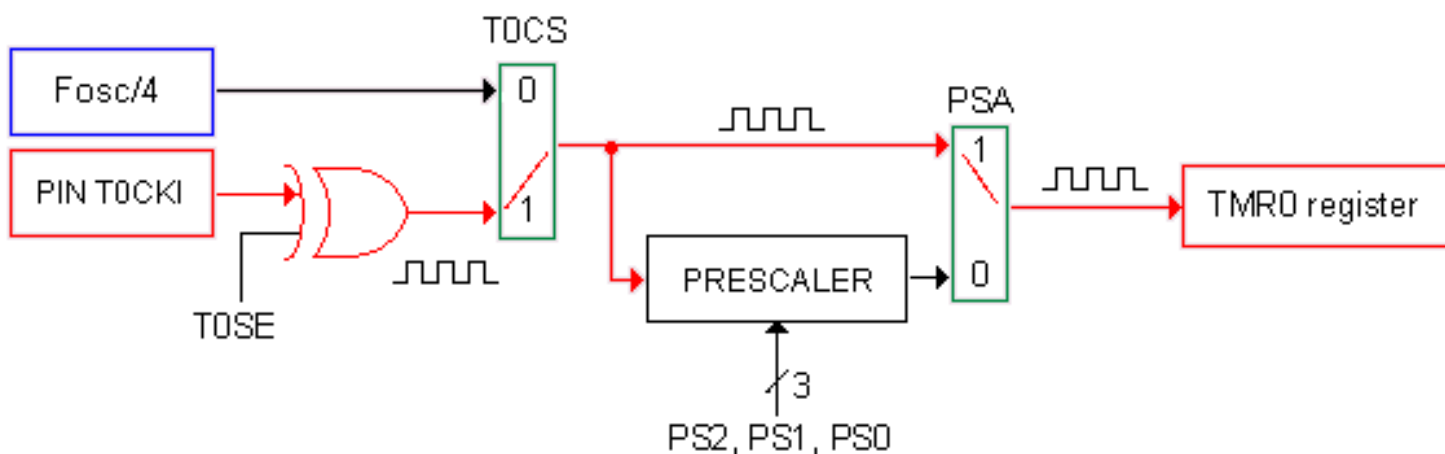
Iniziamo programmando i bit T0CS a 0 e PSA a 1. La configurazione di funzionamento che otterremo è rappresentata nella seguente figura:



Le parti in rosso evidenziano il percorso che effettua il segnale prima di arrivare al contatore TMR0.

Come abbiamo già detto in precedenza, la frequenza Fosc/4 è pari ad un quarto della frequenza di clock. Utilizzando un quarzo da 4Mhz avremo una Fosc/4 pari ad 1 MHz. Tale frequenza viene inviata direttamente al registro TMR0 senza subire nessun cambiamento. La cadenza di conteggio che se ne ottiene è quindi pari ad 1 milione di incrementi al secondo del valore presente in TMR0.

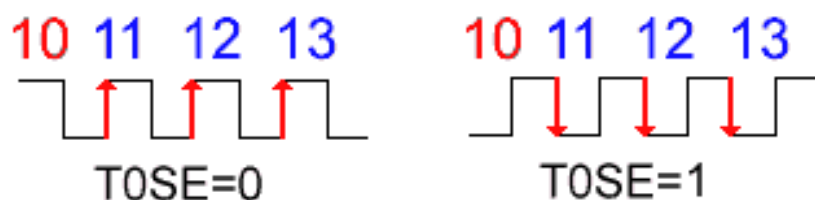
Ipotizziamo ora di cambiare lo stato del bit T0CS da 0 a 1 la configurazione che otteniamo è la seguente:



Questa volta sarà il segnale applicato al pin T0CKI del PIC ad essere inviato direttamente al contatore TMR0 determinandone la frequenza di conteggio. Applicando ad esempio a questo pin una frequenza pari ad 100Hz otterremo una frequenza di conteggio pari a cento incrementi al secondo.

La presenza della porta logica XOR (exclusive OR) all'ingresso TOCKI del PIC consente di determinare tramite il bit T0SE del registro OPTION se il contatore TMR0 deve essere incrementato in corrispondenza del fronte di discesa (T0SE=1) o del fronte di salita (T0SE=0) del segnale applicato dall'esterno.

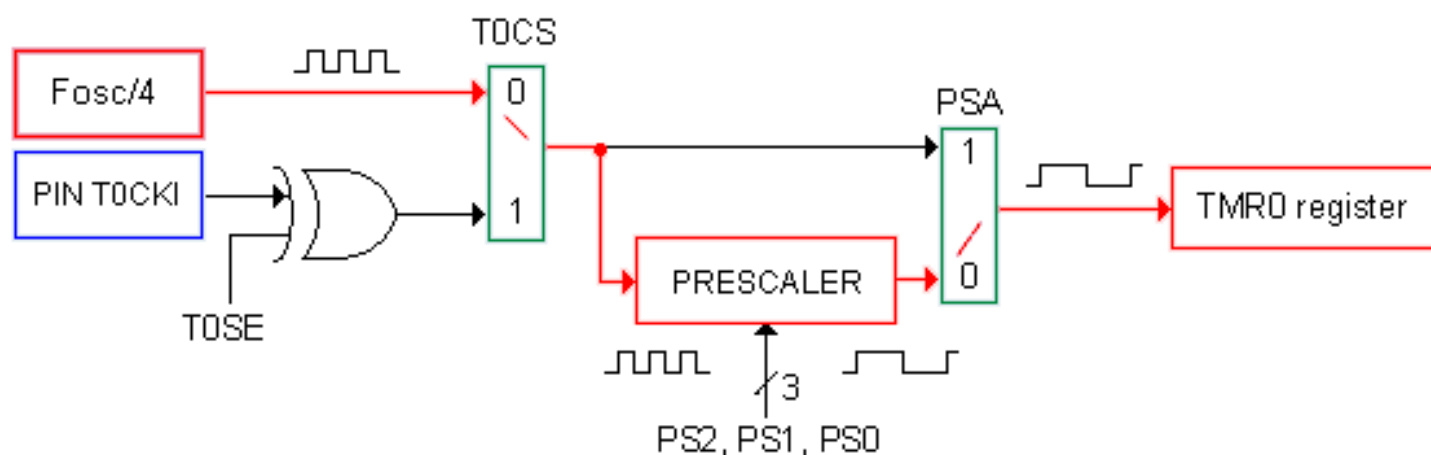
Nella figura seguente viene rappresentata la corrispondenza tra l'andamento del segnale esterno ed il valore assunto dal contatore TMR0 in entrambe i casi:



Il Prescaler

L'ultimo blocco rimasto da analizzare per poter utilizzare completamente il registro TMR0 è il PRESCALER.

Se configuriamo il bit PSA del registro OPTION a 0 inviamo al registro TMR0 il segnale in uscita dal PRESCALER come visibile nella seguente figura:



Il PRESCALER consiste in pratica in un divisore programmabile a 8 bit da utilizzare nel caso la frequenza di conteggio inviata al contatore TMR0 sia troppo elevata per i nostri scopi.

Nell'esempio riportato al passo precedente abbiamo visto che utilizzando un quarzo da 4Mhz otteniamo una frequenza di conteggio pari ad 1Mhz che per molte applicazioni potrebbe risultare troppo elevata.

Con l'uso del PRESCALER possiamo dividere ulteriormente la frequenza Fosc/4 configurando opportunamente i bit PS0, PS1 e PS2 del registro OPTION secondo la seguente tabella.

PS2	PS1	PS0	Divisore	Frequenza in uscita al prescaler (Hz)
0	0	0	2	500.000
0	0	1	4	250.000
0	1	0	8	125.000
0	1	1	16	62.500
1	0	0	32	31.250
1	0	1	64	15.625
1	1	0	128	7.813
1	1	1	256	3.906

Proviamo ora ad effettuare un esperimento sul campo per verificare quanto finora appreso.

Nella lezione 2 avevamo realizzato un lampeggiatore a quattro led la cui sequenza di lampeggio era determinata da una subroutine che introduceva un ritardo software, ovvero un ritardo basato sul tempo di esecuzione di un ciclo continuo di istruzioni.

Proviamo ora a riscrivere la stessa subroutine per introdurre un ritardo pari ad un secondo utilizzando il registro TMR0.

Le modifiche sono state riportate nel file [SEQTMR0.ASM](#).

Dobbiamo anzitutto programmare il PRESCALER per ottenere una frequenza di conteggio conveniente inserendo le seguenti istruzioni all'inizio del programma:

```
movlw    00000100B
movwf    OPTION_REG
```

In pratica dobbiamo programmare bit T0CS a 0 per selezionare come sorgente del conteggio il clock del PIC, il bit PSA a 0 per assegnare il PRESCALER al registro TRM0 anziché al Watch Dog Timer (di cui tratteremo in seguito) e i bit di configurazione del PRESCALER a 100 per ottenere una frequenza di divisione pari a 1:32.

La frequenza di conteggio che otterremo sul TRM0 sarà pari a:

$$Fosc = 1Mhz / 32 = 31.250 \text{ Hz}$$

La subroutine Delay dovrà utilizzare opportunamente il registro TMR0 per ottenere un ritardo pari ad un secondo. Vediamo come. Le prime istruzioni che vengono eseguite nella Delay sono:

```
movlw    6
movwf    TMR0
```

e

```
movlw    125  
movwf    Count
```

Le prime due memorizzano in TMR0 il valore 6 in modo che il registro TMR0 raggiunga lo zero dopo 250 conteggi ($256 - 6 = 250$) ottenendo così una frequenza di passaggi per lo zero di TMR0 pari a:

$$31.250 / 250 = 125 \text{ Hz}$$

Le istruzioni seguenti memorizzano in un registro a 8 bit (Count) il valore 125 in modo tale che, decrementando questo registro di uno per ogni passaggio per lo zero di TMR0, si ottenga una frequenza di passaggi per lo zero del registro Count pari a:

$$125/125 = 1\text{Hz}$$

Le istruzioni inserite nel loop DelayLoop si occupano quindi di controllare se TMR0 ha raggiunto lo zero, quindi di reinizializzarlo a 6 e decrementare il valore contenuto in Count. Quando Count raggiungerà anch'esso lo zero allora sarà trascorso un secondo e la subroutine potrà fare ritorno al programma chiamante.

```

;*****
; Pic by example
; SEQTMR0.ASM
; Luci sequenziali con temporizzazione via TIMER 0
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"

;Setup of PIC configuration flags

;XT oscillator
;Disable watch dog timer
;Enable power up timer
;Disable code protect

__CONFIG      3FF1H

ORG      0CH

Count      RES      1
Shift      RES      1

;Reset Vector - Punto di inizio del programma al reset della CPU

ORG      00H

;Commuta sul secondo banco dei registri

bsf      STATUS,RP0

;Definizione delle linee di I/O (0=Uscita, 1=Ingresso)

movlw    00011111B
movwf    TRISA & 7FH

movlw    11110000B
movwf    TRISB & 7FH

;Assegna il PRESCALER a TMR0 e lo configura a 1:32
;Vedi subroutine Delay per maggiori chiarimenti

movlw    00000100B
movwf    OPTION_REG & 0x7F

;Commuta sul primo banco dei registri

bcf      STATUS,RP0

;Il registro Shift viene utilizzato per rappresentare internamente
;lo stato delle linee di uscita della porta B dove sono collegati i led.
;Il bit 0 del registro Shift viene settato ad uno per iniziare il ciclo
;dal primo led.

movlw    00000001B
movwf    Shift

;Loop di scorrimento
MainLoop

;Invia sulla porta B il registro Shift cosi che ogni bit settato ad uno in Shift
;fara' accendere il led relativo

movf     Shift,W

```

```
movwf    PORTB
```

```

;Per ruotare le luci usa l'istruzione rlf che effettua lo shift a sinistra dei bit
;contenuti nel registro ed inserisce nel bit 0 lo stato del bit di carry. Per
;questo motivo prima di effettuare l'istruzione rlf azzerà il bit di carry con
l'istruzione

```

```
;bcf STATUS,C.
```

```
bcf      STATUS,C
```

```
rlf      Shift,F
```

```

;Quando lo shift raggiunge il bit 4 vengono invertiti i primi quattro bit del registro
;Shift con i secondi quattro bit in modo da ricominciare il ciclo dal bit 0.

```

```
;
```

```
; Ecco cosa succede ai bit del registro Shift durante l'esecuzione di questo loop:
```

```
;
```

```
; 00000001 <--- Valore iniziale (primo led acceso)
```

```
; 00000010 rlf
```

```
; 00000100 rlf
```

```
; 00001000 rlf
```

```
; 00010000 rlf a questo punto viene eseguita l'istruzione swapf ottenendo:
```

```
; 00000001 ...e così' via
```

```
btfsc    Shift,4
```

```
swapf    Shift,F
```

```
;Inserisce un ritardo tra una accensione e l'altra
```

```
call     Delay
```

```
;Torna ad eseguire nuovamente il loop
```

```
goto     MainLoop
```

```
;
```

```
; Subroutines
```

```
;
```

```
; Inserimento di un ritardo pari ad un secondo
```

```
; utilizzando il registro TMR0
```

```
;
```

```
; Il ritardo viene ottenuto dalla frequenza in uscita al PRESCALER pari a:
```

```
; 4Mhz / 4 / 32 = 31.250 Hz
```

```
; ... divisa per 250 dal TMR0 31.250 / 250 = 125 Hz
```

```
; ... e per 125 dal contatore Count 125 / 125 = 1Hz
```

Delay

```
; Inizializza TMR0 per ottenere 250 conteggi prima di arrivare a zero.
```

```
; Il registro TMR0 e' un registro ad 8 bit quindi se viene incrementato
```

```
; nuovamente quando arriva a 255 ricomincia a contare da zero.
```

```
; Se lo si inizializza a 6 dovrà' essere incrementato 256 - 6 = 250 volte
```

```
; prima passare per lo zero.
```

```
movlw    6
```

```
movwf    TMR0
```

```
; Il registro Count viene inizializzato a 125 in quanto il suo scopo e' far
```

```
; uscire il loop
```

```
movlw    125
```

```
movwf    Count
```

;Loop di conteggio

DelayLoop

;TMR0 vale 0 ?

movf TMR0,W

btfss STATUS,Z

goto DelayLoop ;No, aspetta...

movlw 6 ;Si, reimposta TMR0 e controlla se

movwf TMR0 ;e' passato per 125 volte per lo zero

decfsz Count,1

goto DelayLoop

return

END



PIC by example

Gli interrupt

L'interrupt è una particolare caratteristica dei PICmicro (e dei microprocessori in generale) che consente di intercettare un evento esterno, interrompere momentaneamente il programma in corso, eseguire una porzione di programma specializzata per la gestione dell'evento verificatosi e riprendere l'esecuzione del programma principale.

Volendo fare un paragone con il mondo reale possiamo dire che l'interrupt rappresenta per il PIC quello che per noi rappresenta ad esempio la suoneria del telefono.

Per poter ricevere telefonate non dobbiamo preoccuparci di alzare continuamente la cornetta per vedere se c'è qualcuno che vuol parlare con noi, ma, grazie alla suoneria, possiamo continuare tranquillamente a fare le nostre faccende in quanto saremo avvisati da questa ogni volta che qualcuno ci sta chiamando.

Appena sentiamo lo squillo, possiamo decidere di interrompere momentaneamente le nostre faccende, rispondere al telefono e, una volta terminata la conversazione, riprendere dal punto in cui avevamo interrotto.

Riportando i termini di questo paragone al PIC abbiamo che:

- le nostre faccende corrispondono al programma in esecuzione;
- la chiamata da parte di qualcuno corrisponde all'evento da gestire;
- lo squillo del telefono corrisponde alla richiesta di interrupt;
- la nostra risposta al telefono corrisponde alla subroutine di gestione dell'interrupt.

E' evidente quanto sia più efficiente gestire un evento con un interrupt anzichè controllare ciclicamente il verificarsi dell'evento con il programma principale. Gran parte degli aspetti legati alla gestione dell'interrupt vengono inoltre trattati direttamente dall'hardware interno del PIC per cui il tempo di risposta all'evento è praticamente immediato.

Tipi di evento e bit di abilitazione

Il PIC16F84A è in grado di gestire in interrupt quattro eventi diversi, vediamo quali sono:

1. Il cambiamento di stato sulla linea RB0 (External interrupt RB0/INT pin).
2. La fine del conteggio del registro TMR0 (TMR0 overflow interrupt).
3. Il cambiamento di stato su una delle linee da RB4 ad RB7 (PORTB change interrupts).
4. La fine della scrittura su una locazione EEPROM (EEPROM write complete interrupt).

L'interrupt su ognuno di questi eventi può essere abilitato o disabilitato indipendentemente dagli altri agendo sui seguenti bit del registro INTCON:

- INTE (bit 4) se questo bit viene messo a 1 viene abilitato l'interrupt sul

cambiamento di stato sulla linea RB0

- T0IE (bit 5) se questo bit viene messo a 1 viene abilitato l'interrupt sulla fine del conteggio del registro TMR0
- RBIE (bit 3) se questo bit viene messo a 1 viene abilitato l'interrupt sul cambiamento di stato su una delle linee da RB4 ad RB7
- EEIE (bit 6) se questo bit viene messo a 1 viene abilitato l'interrupt sulla fine della scrittura su una locazione EEPROM

Esiste inoltre un bit di abilitazione generale degli interrupt che deve essere settato anch'esso ad uno ovvero il bit GIE (Global Interrupt Enable bit) posto sul bit 7 del registro INTCON.

Interrupt vector ed Interrupt handler

Qualunque sia l'evento abilitato, al suo manifestarsi il PIC interrompe l'esecuzione del programma in corso, memorizza automaticamente nello STACK il valore corrente del PROGRAM COUNTER e salta all'istruzione presente nella locazione di memoria 0004H denominata Interrupt vector (vettore di interrupt).

E' da questo punto che dobbiamo inserire la nostra subroutine di gestione dell'interrupt denominata Interrupt Handler (gestore di interrupt).

Potendo abilitare più interrupt, tra i primi compiti dell'interrupt handler è la verifica di quale, tra gli eventi abilitati, ha generato l'interrupt e l'esecuzione della parte di programma relativo.

Questo controllo può essere effettuato utilizzando gli Interrupt flag.

Interrupt flag

Dato che qualunque interrupt genera una chiamata alla locazione 04H, nel registro INTCON sono presenti dei flag che indicano quale è l'evento che ha generato l'interrupt, vediamo:

- INTF (bit 1) Se vale 1 l'interrupt è stato generato dal cambiamento di stato sulla linea RB0.
- T0IF (bit 2) Se vale 1 l'interrupt è stato generato al termine del conteggio del timer TMR0.
- RBIF (bit 0) Se vale 1 l'interrupt è stato generato dal cambiamento di stato di una delle linee da RB4 a RB7.

Come si vede per l'interrupt sul fine scrittura in EEPROM non è previsto alcun flag di segnalazione per cui l'interrupt handler dovrà considerare che l'interrupt è stato generato da questo evento quando tutti e tre i flag sopra citati valgono 0.

Importante: Una volta rilevato quale flag è attivo, l'interrupt handler deve azzerarlo altrimenti non verrà più generato l'interrupt corrispondente.

Ritorno da un interrupt handler

Quando viene generato un interrupt il PIC disabilita automaticamente il bit GIE (Global Interrupt Enable) del registro INTCON in modo da disabilitare tutti gli interrupt mentre è già in esecuzione un interrupt handler. Per poter ritornare al programma principale e reinizializzare a 1 questo bit occorre utilizzare l'istruzione:

RETFIE

Esempio pratico di gestione di un interrupt

Vediamo ora un esempio pratico di gestione degli interrupt. Prendiamo come base di partenza il source LED.ASM usato nella lezione 1 per realizzare il lampeggiatore a led. Come ricorderete questo programma fa semplicemente lampeggiare il LED1 a ciclo continuo utilizzando un ritardo software introdotto dalla subroutine Delay.

Vediamo ora come è possibile fargli rilevare la pressione di un tasto ed accendere il LED 2 contemporaneamente all'esecuzione del programma principale.

Il source d'esempio che andremo ad analizzare è disponibile nel file [INTRB.ASM](#)

Proviamo a compilarlo ed a eseguirlo utilizzando lo stesso schema elettrico realizzato nella lezione 3 (file example3.pdf in formato Acrobat Reader 12Kb).

Una volta scaricato il programma INTRB.ASM nella scheda PicTech noteremo che il LED 1 lampeggia esattamente come avveniva con il programma LED.ASM. Proviamo ora a premere uno qualsiasi dei tasti da SW1 a SW4 e vedremo che il LED 2 si accende immediatamente e rimane acceso per un tempo pari a 3 lampeggi del LED 1.

In pratica mentre il loop principale, derivato dal vecchio LED.ASM, continua a far lampeggiare il LED 1 utilizzando un ritardo software introdotto dalla subroutine Delay, il PIC è in grado di accorgersi della pressione di un tasto e di segnalarlo immediatamente sul LED 2 senza influenzare in maniera evidente la frequenza di lampeggio di LED1.

Prima di analizzare il source INTRB.ASM vediamo la differenza di comportamento con un altro source che effettua le stesse operazioni ma senza ricorrere agli interrupt.

A questo proposito compiliamo ed inseriamo nel PICmicro il programma NOINTRB.ASM. Noteremo che l'accensione del LED 2, in corrispondenza alla pressione di un tasto, è leggermente ritardata in quanto la lettura dello stato delle linee RB4-7 non viene effettuata dall'hardware di gestione dell'interrupt ma direttamente dal programma principale ad ogni ciclo di loop. Il leggero ritardo è quindi dovuto alla presenza della subroutine Delay all'interno del loop principale.

Analizziamo ora il source INTRB.ASM

Partiamo dalla direttiva ORG 00H che, come sappiamo serve a posizionare il nostro programma a partire dalla locazione di reset, ovvero dalla locazione con indirizzo 0.

Notiamo subito che la prima istruzione che incontra il PIC è un salto incondizionato alla label Start:

```
ORG 0x00
goto Start
```

seguito da un'altra direttiva:

```
ORG 0x04
```

e quindi dal codice della subroutine di gestione dell'interrupt:

```
bsf PORTB,LED2

movlw 3
movwf nTick

bcf INTCON,RBIF

retfie
```

Come abbiamo detto nella lezione precedente, l'interrupt handler deve necessariamente essere allocato a partire dall'indirizzo 0x04, quindi per evitare che venga eseguito al reset dobbiamo necessariamente saltarlo con una istruzione di salto incondizionato.

Il codice dell'interrupt handler, in questo caso, è molto semplice e si limita ad accendere il LED 2, quindi inserire nel registro utente nTick il numero di lampeggi raggiunto il quale il LED 2 deve spegnersi e quindi azzerare il flag RBIF per consentire alla circuiteria di generazione dell'interrupt di continuare a funzionare.

L'istruzione RETFIE consente al PIC di tornare ad eseguire il programma interrotto dall'interrupt.

Ma perchè viene generato un interrupt quando premiamo un tasto qualsiasi ?

Tra le prime istruzioni che esegue il nostro PIC al reset troviamo le seguenti:

```
movlw 10001000B
movwf INTCON
```

dove in pratica viene messo ad uno il bit GIE (bit 7) che abilita in generale la circuiteria di generazione degli interrupt e quindi il bit RBIE (bit 3) che abilita, in particolare, l'interrupt su cambiamento di stato delle linee RB4-7.

In pratica, avendo collegato i pulsanti PU1, PU2, PU3 e PU4 proprio sulle linee di I/O RB4, RB5, RB6 ed RB7, con la pressione di uno di questi otteniamo un cambiamento di stato e quindi un interrupt.

Nel loop principale, oltre alle operazioni di accensione e spegnimento del LED 1, viene decrementato il contatore nTick fino al raggiungimento dello zero. In corrispondenza di questo viene spento il LED 2.

Esempio pratico di gestione di più interrupt

Vediamo ora come gestire più interrupt contemporaneamente.

Utilizzando sempre come base il source precedente INTRB.ASM proviamo a gestire anche l'interrupt sulla fine conteggio del registro TMR0. Ovvero facciamo lampeggiare il LED 3 in corrispondenza di ogni fine conteggio di TMR0.

Il source da utilizzare è [DBLINT.ASM](#).

Compiliamo e scarichiamo il programma DBLINT.ASM nella scheda PicTech e vediamo che oltre a LED 1 che lampeggia con la solita frequenza, c'è il LED 3 che lampeggia contemporaneamente con una frequenza più elevata.

Premendo un tasto qualsiasi, inoltre, otteniamo la solita accensione per tre cicli del LED 2. L'effetto finale che otteniamo è l'esecuzione di tre compiti ad una velocità tale da sembrare in esecuzione parallela.

Analizziamo ora il source DBLINT.ASM

Le modifiche maggiori riguardano l'interrupt handler all'inizio del quale viene effettuato un controllo su quale evento abbia generato l'interrupt. Con le istruzioni:

```

btfsc  INTCON,T0IF
goto   IntT0IF
btfsc  INTCON,RBIF
goto   IntRBIF

```

viene controllato il flag T0IF e RBIF per vedere rispettivamente se l'evento che ha scatenato l'interrupt proviene dal registro TMR0 o dalle porta RB4-RB7. Quindi vengono lanciate in esecuzione le relative subroutine di gestione a partire dalle label IntT0IF e IntRBIF.

Prima di ridare il controllo al programma principale vengono azzerati i flag T0IF e RBIF assicurarsi che i successivi eventi possano scaturire nuovamente gli interrupt.

```

;*****
; Pic by example
; INTRB.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"

;Setup of PIC configuration flags

;XT oscillator
;Disable watch dog timer
;Enable power up timer
;Disable code protect

__CONFIG      3FF1H

LED1 EQU    0
LED2 EQU    1
LED3 EQU    2
LED4 EQU    3

ORG      0CH

Count RES    2
nTick  RES    1      ;Registro utilizzato per contare il numero di
                    ;lampeggi del LED 1

;Reset Vector
;Punto di inizio del programma al reset della CPU

ORG      00H

;Salta al corpo principale del programma. Questo jump   necessario
;per evitare tutta la parte di codice per la gestione degli
;interrupt.

goto     Start

;Interrupt vector
;Punto di inizio per tutte le subroutine di gestione degli interrupt

ORG      04H

;*****
; Interrupt handler
;*****

;Accende il led 2 per segnalare che c'e' stato un interrupt
bsf      PORTB,LED2

;Inizializza il contatore di lampeggi del LED1
movlw    3
movwf    nTick

;Azzerare nuovamente il flag RBIF per consentire all'interrupt di
;ripetersi
bcf      INTCON,RBIF

;Ritorna al programma principale
retfie

;*****
; Programma principale

```

```
;*****
```

Start:

```
;Commuta sul secondo banco dei registri per accedere ai registri TRISA e TRISB
```

```
bsf      STATUS,RP0
```

```
;Definizione delle linee di I/O (0=Uscita, 1=Ingresso)
```

```
;Definizione della porta A
```

```
movlw    00011111B
```

```
movwf    TRISA & 7FH
```

```
;Definizione della porta B
```

```
;Le linee da RB0 a RB3 vengono programmate in uscita per essere collegate ai quattro
```

led

```
;Le linee da RB4 a RB7 vengono programmate in ingresso per essere collegate ai quattro
```

pulsanti

```
movlw    11110000B
```

```
movwf    TRISB & 7FH
```

```
;Commuta sul primo banco dei registri
```

```
bcf      STATUS,RP0
```

```
;Spegne tutti i led collegati sulla porta B
```

```
bcf      PORTB,LED1
```

```
bcf      PORTB,LED2
```

```
bcf      PORTB,LED3
```

```
bcf      PORTB,LED4
```

```
;Abilita l'interrupt sul cambiamento di stato delle linee RB4,5,6,7
```

```
movlw    10001000B
```

```
movwf    INTCON
```

```
;*****
```

```
; Loop principale
```

```
;*****
```

MainLoop

```
call     Delay                ;Ritardo software
```

```
btfss    PORTB,LED1           ;Led acceso ?
```

```
goto     TurnOnLed1           ;No, lo accende
```

```
goto     TurnOffLed1          ;Si, lo spegne
```

```
;Accensione led e decremento del contatore di lampeggi
```

TurnOnLed1

```
bsf      PORTB,LED1
```

```
;Controlla se LED 2 di segnalazione dell'interrupt e' gia acceso.
```

```
;Se e' acceso decrementa il contatore nTick ad ogni lampeggio di
```

```
;LED1. Quando nTick vale 0 spegne LED 2
```

```
btfss    PORTB,LED2           ;LED2 acceso ?
```

```
goto     MainLoop            ;No, continua a lampeggiare
```

```
decf     nTick,1              ;Si, decrementa nTick
```

```
btfss    STATUS,Z             ;nTick = 0 ?
```

```
goto     MainLoop            ;No, continua a lampeggiare
```

```
bcf      PORTB,LED2           ;Si, spegne LED2
```

```
goto     MainLoop            ;Continua a lampeggiare
```

```
;Spegnimento led
```

```
TurnOffLed1
```

```
bcf    PORTB,LED1    ;Spegne LED 1
goto   MainLoop      ;Continua a lampeggiare
```

```
;*****
;
; Subroutine
;*****
```

```
;Subroutine di ritardo software
```

```
Delay
```

```
clrf   Count
clrf   Count+1
```

```
DelayLoop
```

```
decfsz Count,1
goto   DelayLoop
```

```
decfsz Count+1,1
goto   DelayLoop
```

```
return
```

```
END
```

```

;*****
; Pic by example
; DBLINT.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"
        ERRORLEVEL     -302

;Setup of PIC configuration flags

;XT oscillator
;Disable watch dog timer
;Enable power up timer
;Disable code protect

__CONFIG      3FF1H

LED1 EQU      0
LED2 EQU      1
LED3 EQU      2
LED4 EQU      3

ORG      0CH

Count RES     2
nTick  RES     1

;Reset Vector
;Starting point at CPU reset

ORG      00H

;Jump to the main body of program to avoid the interrupt handler
;code.

goto     Start

;Interrupt vector
;Starting point at CPU interrupts

ORG      04H

;*****
; Interrupt handler
;*****

;Check the interrupt event

btfsc    INTCON,T0IF
goto     IntT0IF
btfsc    INTCON,RBIF
goto     IntrBIF

;Reset the T0IF and RBIF flags to re-enable the interrupts

End_ih   bcf      INTCON,T0IF
         bcf      INTCON,RBIF

;Go back to the main program
retfie

;*****
; TMR0 Interrupt handler

```

```
;*****
```

IntT0IF

```
;Turn on LED3 if it's off
```

```
btfsc    PORTB, LED3
goto     LED3_off
```

LED3_off

```
bsf      PORTB, LED3
goto     End_ih
```

```
bcf      PORTB, LED3
goto     End_ih
```

```
;*****
; RB4-RB7 interrupt handler
;*****
```

IntrBIF

```
;Turn on LED 2
```

```
bsf      PORTB, LED2
```

```
;Starts the LED1 blink counter
```

```
movlw    3
movwf    nTick
```

```
goto     End_ih
```

```
;*****
; Main body
;*****
```

Start:

```
;Commuta sul secondo banco dei registri per accedere ai registri TRISA e TRISB
```

```
bsf      STATUS, RP0
```

```
;Definizione delle linee di I/O (0=Uscita, 1=Ingresso)
;Definizione della porta A
```

```
movlw    00011111B
movwf    TRISA & 7FH
```

```
;Definizione della porta B
```

```
;Le linee da RB0 a RB3 vengono programmate in uscita per essere collegate ai quattro
```

led

```
;Le linee da RB4 a RB7 vengono programmate in ingresso per essere collegate ai quattro
```

pulsanti

```
movlw    11110000B
movwf    TRISB & 7FH
```

```
;Assegna il PRESCALER a TMR0 e lo configura a 1:256
```

```
movlw    00000111B
movwf    OPTION_REG & 7FH
```

```
;Commuta sul primo banco dei registri
```

```
bcf      STATUS, RP0
```

```
;Spegne tutti i led collegati sulla porta B
```

```
bcf      PORTB, LED1
bcf      PORTB, LED2
bcf      PORTB, LED3
bcf      PORTB, LED4
```

```
;Abilita l'interrupt sul TMR0 e sul cambiamento di stato delle linee RB4,5,6,7
```

```
movlw    10101000B
movwf    INTCON
```

```
;*****
;
; Loop principale
;*****
```

MainLoop

```
call     Delay                ;Ritardo software

btfss    PORTB,LED1           ;Led acceso ?
goto     TurnOnLed1           ;No, lo accende
goto     TurnOffLed1          ;Si, lo spegne

;Accensione led e decremento del contatore di lampeggi
```

TurnOnLed1

```
bsf       PORTB,LED1

;Controlla se LED 2 di segnalazione dell'interrupt e' gia acceso.
;Se e' acceso decrementa il contatore nTick ad ogni lampeggio di
;LED1. Quando nTick vale 0 spegne LED 2

btfss    PORTB,LED2           ;LED2 acceso ?
goto     MainLoop             ;No, continua a lampeggiare

decf     nTick,1              ;Si, decrementa nTick
btfss    STATUS,Z             ;nTick = 0 ?
goto     MainLoop             ;No, continua a lampeggiare

bcf       PORTB,LED2          ;Si, spegne LED2
goto     MainLoop             ;Continua a lampeggiare

;Spegnimento led
```

TurnOffLed1

```
bcf       PORTB,LED1          ;Spegne LED 1
goto     MainLoop             ;Continua a lampeggiare
```

```
;*****
;
; Subroutine
;*****
```

```
;Subroutine di ritardo software
```

Delay

```
clrf     Count
clrf     Count+1
```

DelayLoop

```
decfsz   Count,1
goto     DelayLoop

decfsz   Count+1,1
goto     DelayLoop

return

END
```



PIC by example

Il Power Down Mode

Il Power Down Mode o Sleep Mode è un particolare stato di funzionamento del PICmicro utilizzato per ridurre il consumo di corrente nei momenti in cui il PICmicro non è utilizzato perchè in attesa di un evento esterno.

Se prendiamo come esempio un telecomando per apricancello o per TV vediamo che per la maggior parte del tempo il PICmicro rimane in attesa che qualcuno prema un tasto. Appena premuto il PICmicro effettua una breve trasmissione e si rimette di nuovo in attesa della pressione del prossimo tasto.

Il tempo di utilizzo effettivo della CPU del PICmicro è quindi limitato ai pochi millisecondi necessari per effettuare la trasmissione mentre per diverse ore non è richiesta nessuna elaborazione particolare.

Per evitare di consumare inutilmente la limitata energia dalla batteria è possibile spegnere buona parte dei circuiti di funzionamento del PICmicro e riaccenderli solo in corrispondenza di un qualche evento esterno.

Vediamo come.

L'istruzione SLEEP

L'istruzione SLEEP viene utilizzata per mettere il PIC in Power Down Mode e ridurre di conseguenza la corrente assorbita che passerà da circa 2mA (a 5 volt con clock di funzionamento a 4Mhz) a circa 2µA, ovvero 1000 volte di meno !

Per entrare in Power Down Mode basta inserire questa istruzione in un punto qualsiasi del nostro programma:

SLEEP

Qualsiasi istruzione presente dopo la SLEEP non verrà eseguita dal PICmicro che terminerà in questo punto la sua esecuzione, spegnerà tutti i circuiti interni, tranne quelli necessari a mantenere lo stato delle porte di I/O (stato logico alto, basso o alta impedenza) ed a rilevare le condizioni di "risveglio" di cui parleremo di seguito.

Per ridurre il consumo di corrente in questo stato, non devono esserci ovviamente circuiti collegati alle linee di uscita del PIC che consumino corrente. O meglio questi circuiti devono essere progettati in modo da limitare il loro assorbimento nelle condizioni di Power Down. Un altro accorgimento raccomandato dalla Microchip è quello di collegare al positivo (Vdd) o al negativo (Vss) di alimentazione tutte le linee in alta impedenza non utilizzate compresa la linea RA4/T0CKI (pin 3).

Il "risveglio" del PICmicro

Per risvegliare il PICmicro dal suo sonno possono essere utilizzate diverse tecniche:

1. Reset del PICmicro mettendo a 0 il pin MCLR (pin 4)
2. Timeout del timer del Watchdog (se abilitato)
3. Verificarsi di una situazione di interrupt (interrupt dal pin RB0/INT, cambio di stato sulla porta B, termine delle operazioni di scrittura su EEPROM)

Nei casi 1 e 2 il PICmicro viene resettato e l'esecuzione ripresa dalla locazione 0.

Nel caso 3 il PICmicro si comporta come nella normale gestione di un interrupt eseguendo per primo l'interrupt handler e quindi riprendendo l'esecuzione dopo l'istruzione SLEEP. Perché il PICmicro venga risvegliato da un interrupt devono essere abilitati opportunamente i flag del registro INTCON.

Esempio di Power Down mode

Vediamo ora un primo semplice esempio di utilizzo del Power Down Mode e di modalità di "risveglio" del PICmicro. La modalità utilizzata è l'interrupt sul fronte di discesa applicato al pin RB0/INT utilizzando un pulsante. Il source utilizzato è [PDM.ASM](#).

Lo schema da realizzare è disponibile nel seguente file [example4.pdf](#).

In pratica il LED D1 collegato alla linea RB2 lampeggerà ad indicare l'esecuzione del programma in corso. Premendo il tasto SW2 il programma eseguirà l'istruzione:

SLEEP

mettendo il PICmicro in Power Down Mode. Il LED D1 rimarrà acceso o spento in base al momento scelto per premere SW2.

Per causare l'uscita dal Power Down Mode del PICmicro, basterà premere SW1 per generare un interrupt e far riprendere l'esecuzione del programma.

```

; *****
; Pic by example
;
; PDM.ASM
; Power Down Mode example
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
; *****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"

;Set chip configuration

;Disable watch dog timer
;Enable power up timer
;XT oscillator
;Disable code protect

__CONFIG      3FF1H

SWITCH1 EQU      0
SWITCH2 EQU      1
LED1      EQU      2

        ORG      0CH

;16 bit counter used in the delay subroutine

Count      RES      2

;Reset Vector
;Start point at CPU reset

        ORG      00H

;Jump to main body of program.

        goto      Start

; *****
; Interrupt vector
; Start point for every interrupt handler
; *****

        ORG      04H

; *****
; Interrupt handler
; *****

        bcf      INTCON,INTF      ;Reset INTF flag
        retfie      ;Return to the main body

; *****
; Main body
; *****

Start:

        bsf      STATUS,RP0      ;Swap to data bank 1

;I/O lines definition on port A (0=output, 1=input)

        movlw    00011111B      ;Definition of port a
        movwf    TRISA & 0x7F

```

```

;I/O lines definition on port B (0=output, 1=input)

bsf      TRISB & 0x7F, SWITCH1    ;Switch 1
bsf      TRISB & 0x7F, SWITCH2    ;Switch 2
bcf      TRISB & 0x7F, LED1        ;Led 1

;Set to 0 the INTEDG bit on OPTION register
;to have an interrupt on the falling edge of RB0/INT

bcf      OPTION_REG & 0x7F, INTEDG
bcf      STATUS, RP0              ;Swap to data bank 0

bsf      INTCON, GIE               ;Enables interrupts
bsf      INTCON, INTE              ;Enables RB0/INT interrupt

bcf      PORTB, LED1              ;Turn off LED1

;*****
;
; Main loop
;*****

```

MainLoop

```

btfss    PORTB, SWITCH2           ;If switch2 is down enter in
sleep                                         ;Power Down Mode

call     Delay                    ;Software delay

;If LED1 in on then turn it off and viceversa

btfss    PORTB, LED1              ;Led on ?
goto     TurnOnLed1               ;No, turn it on
goto     TurnOffLed1              ;Yes, turn it off

;Turn LED1 on

```

TurnOnLed1

```

bsf      PORTB, LED1
goto     MainLoop

;Turn LED1 off

```

TurnOffLed1

```

bcf      PORTB, LED1
goto     MainLoop

;*****
;
; Software delay
;*****

```

Delay

```

clrf     Count
clrf     Count+1

```

DelayLoop

```

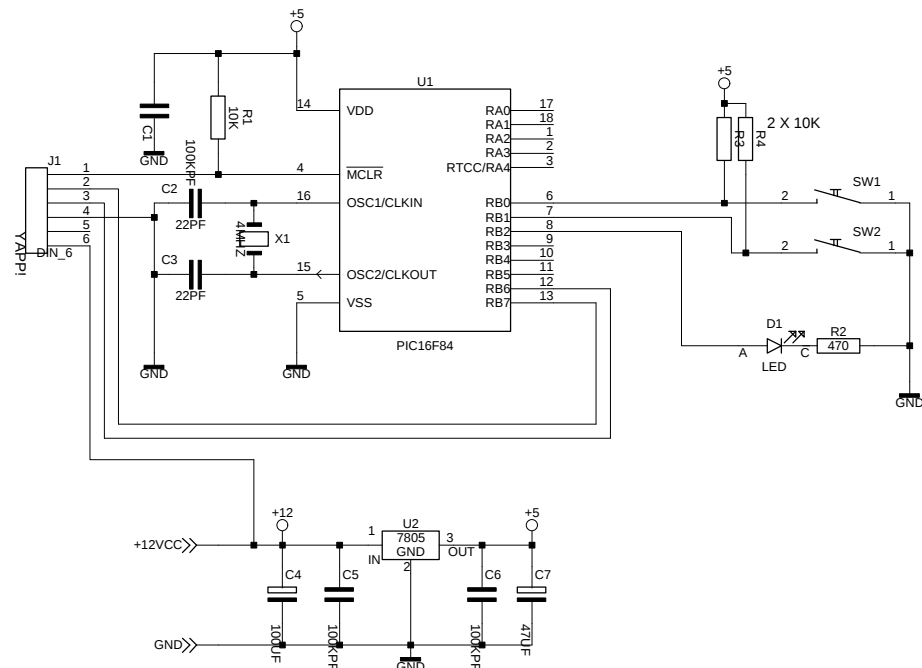
decfsz   Count, 1
goto     DelayLoop

decfsz   Count+1, 1
goto     DelayLoop

return

END

```



WWW.PICPOINT.COM		
PROJECT: PIC BY EXAMPLE		
NOTE: EXAMPLE N.4		
ENGINEER: SERGIO TANZILLI		
VER: 1.0	DATE: 2 NOV 1998	PAGES: 1 OF: 1



PIC by example

Il watch dog timer (WDT)

In questa lezione analizzeremo il funzionamento del Watch Dog Timer (che tradotto in italiano potrebbe significare Temporizzatore Cane da Guardia) il cui scopo è quello di migliorare l'affidabilità dei nostri circuiti basati su PICmicro.

Il Watch Dog Timer è in pratica un oscillatore interno al PICmicro, ma completamente indipendente dal resto della circuiteria, il cui scopo è quello di rilevare eventuali blocchi della CPU del micro e resettare il PICmicro per riprendere la normale esecuzione del programma.

Per poter rilevare un eventuale blocco della CPU durante l'esecuzione del programma principale, viene inserita all'interno di questo, una istruzione speciale, la:

CLRWDT (CLear Watch Dog Timer)

la quale azzerava ad intervalli regolari il Watch Dog Timer non consentendogli di terminare il suo conteggio. Se la CPU non effettua questa istruzione prima del termine del conteggio allora si assume che il programma si è bloccato per qualche motivo e si effettua il Reset della CPU.

Il periodo minimo raggiunto il quale la CPU viene resettata è di circa 18ms (dipende dalla temperatura e dalla tensione di alimentazione). E' possibile però assegnare il PRESCALER al Watch Dog Timer per ottenere ritardi più lunghi fino a 2.3 secondi.

Per abilitare il Watch Dog Timer occorre abilitare in fase di programmazione il flag WDTE della word di configurazione. La modalità di attivazione di questo flag dipende dal programmatore in uso.

Assegnazione del PRESCALER al WDT

Agendo sul bit PSA del registro OPTION_REG è possibile assegnare il prescaler al Watch Dog Timer per ottenere dei tempi di ritardo di intervento maggiori. Il bit PSA va settato ad uno con l'istruzione:

BSF OPTION_REG, PSA

In caso contrario il prescaler verrà assegnato al TIMER 0. Ovviamente assegnando il prescaler al WDT non sarà possibile assegnarlo completamente al TIMER 0 e viceversa.

Intervenendo sui valori dei bit PS0, PS1 e PS2 dello stesso registro OPTION_REG potremmo ottenere diversi intervalli di ritardo. La scelta corretta dovrà essere fatta tenendo conto del massimo ritardo che riusciamo ad ottenere all'interno del nostro programma tra l'esecuzione di due istruzioni CLRWDT successive.

Nella tabella seguente è riportato la corrispondenza tra i valori di questi bit e gli intervalli che otterremo.

PS2	PS1	PS0	Divisore	Periodo di ritardo del WDT
0	0	0	1	18ms
0	0	1	2	36ms
0	1	0	4	72ms
0	1	1	8	144ms
1	0	0	16	288ms
1	0	1	32	576ms
1	1	0	64	1.152s
1	1	1	128	2.304s

Esempio pratico di uso del Watch Dog Timer

Vediamo ora, come sempre, un esempio pratico di utilizzo del Watch Dog Timer. Useremo lo stesso schema usato nell'esempio precedente e riportato nel file [example4.pdf](#), il source è riportato nel file [WDT.ASM](#).

In pratica questo esempio non differisce molto dall'esempio già usato per il Power Down Mode.

In pratica appena il programma entrerà in esecuzione vedremo il LED 1 lampeggiare. Durante il lampeggio viene eseguita continuamente l'istruzione CLRWDT per evitare che la CPU venga resettata (a tale proposito bisogna ricordarsi di programmare il PICmicro con l'opzione WDTE abilitata).

Non appena premiamo il tasto SW2 la CPU entra in un loop infinito (StopLoop) senza eseguire la CLRWDT.

Trascorsi circa 2.3 secondi, il Watch Dog Timer effettua il reset della CPU ed il led comincia nuovamente a lampeggiare.

Proviamo ora a riprogrammare il PIC16F84A con lo stesso programma ma senza abilitare il WDTE con il nostro programmatore. Noterete che premendo il tasto SW2 il lampeggio si blocca e non si sblocca più.

```

;*****
; Pic by example
; WDT.ASM
; Watch Dog Timer example
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"

;Setup of chip flags

;Enable watch dog timer
;Enable power up timer
;XT oscillator
;Disable code protect

__CONFIG      3FF5H

SWITCH1 EQU      0
SWITCH2 EQU      1
LED1      EQU      2

        ORG      0CH

;16 bit counter used in the delay subroutine

Count      RES      2

;Reset Vector
;Start point at CPU reset

        ORG      00H

;Jump to main body of program.

goto      Start

;*****
; Interrupt vector
; Start point for every interrupt handler
;*****

        ORG      04H

;*****
; Interrupt handler
;*****

        bcf      INTCON,INTF      ;Reset INTF flag
        retfie      ;Return to the main body

;*****
; Main body
;*****

Start:

        bsf      STATUS,RP0      ;Swap to data bank 1

;I/O lines definition on port A (0=output, 1=input)

        movlw    00011111B      ;Definition of port a
        movwf    TRISA & 0x7F

;I/O lines definition on port B (0=output, 1=input)

```

```

bsf      TRISB & 0x7F, SWITCH1    ;Switch 1
bsf      TRISB & 0x7F, SWITCH2    ;Switch 2
bcf      TRISB & 0x7F, LED1        ;Led 1

;Set to 0 the INTEDG bit on OPTION register
;to have an interrupt on the falling edge of RB0/INT

bcf      OPTION_REG & 0x7F, INTEDG

;Assign the PRESCALER to Watch dog timer

bsf      OPTION_REG & 0x7F, PSA

;Set the PRESCALER to 1:128

bsf      OPTION_REG & 0x7F, PS0
bsf      OPTION_REG & 0x7F, PS1
bsf      OPTION_REG & 0x7F, PS2

bcf      STATUS, RP0              ;Swap to data bank 0

bsf      INTCON, GIE              ;Enables interrupts
bsf      INTCON, INTE             ;Enables RB0/INT interrupt

bcf      PORTB, LED1              ;Turn off LED1

;*****
;
; Main loop
;*****

```

MainLoop

```

btfss    PORTB, SWITCH2          ;If switch2 is down enter in
StopLoop
goto     StopLoop                ;Stops CPU

clrwdt                                ;Clear wtdog timer

call     Delay                    ;Software delay

;If LED1 is on then turn it off and viceversa

btfss    PORTB, LED1             ;Led on ?
goto     TurnOnLed1              ;No, turn it on
goto     TurnOffLed1             ;Yes, turn it off

;Turn LED1 on

```

TurnOnLed1

```

bsf      PORTB, LED1
goto     MainLoop

;Turn LED1 off

```

TurnOffLed1

```

bcf      PORTB, LED1
goto     MainLoop

```

```

;*****
;
; Software delay
;*****

```

Delay

```

clrf     Count
clrf     Count+1

```

DelayLoop

```
decfsz    Count,1  
goto      DelayLoop  
  
decfsz    Count+1,1  
goto      DelayLoop  
  
return  
  
END
```

PIC by example

Gestione di un display LCD

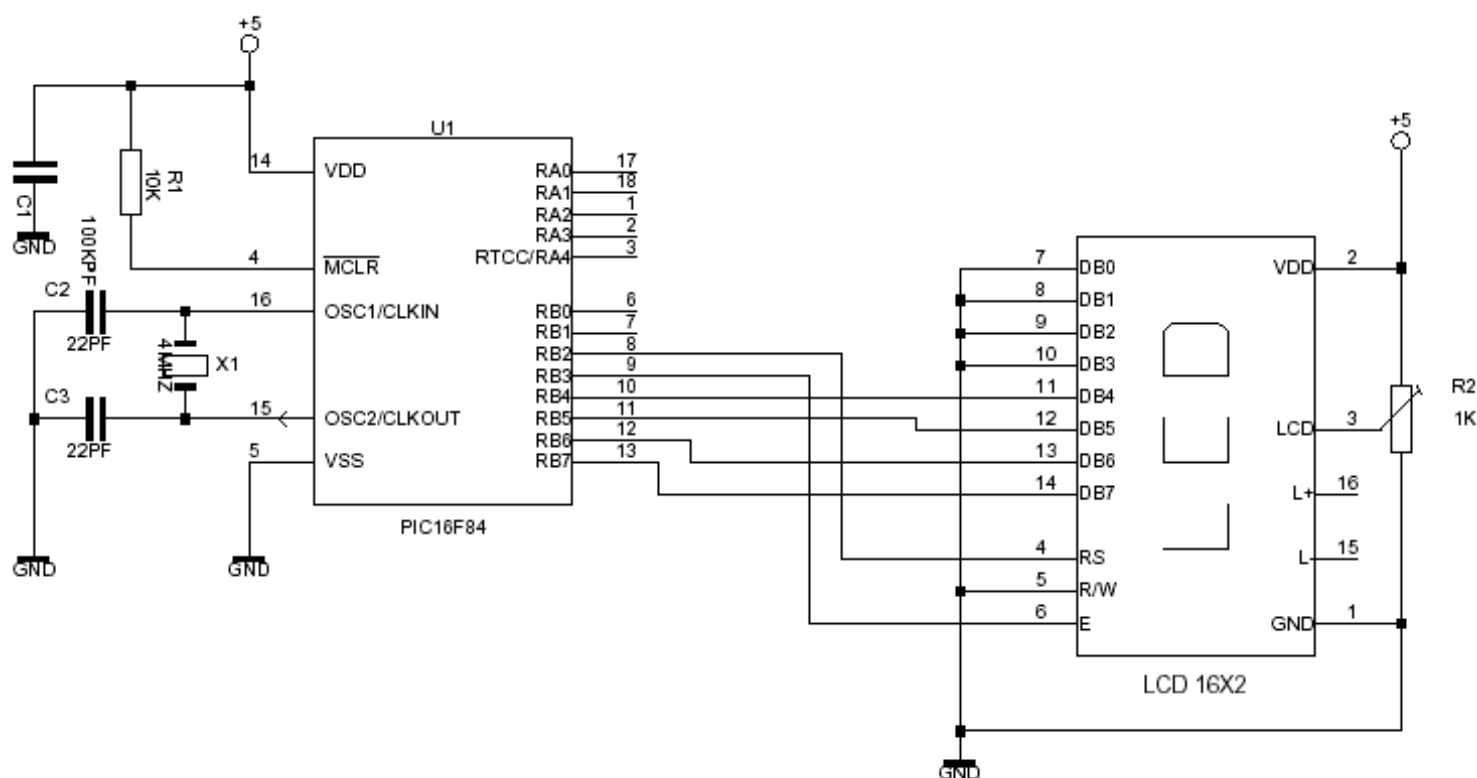
Dopo aver realizzato nelle lezioni precedenti dei semplici esperimenti con diodi led e pulsanti, iniziamo da questa lezione ad interfacciare il nostro PIC16F84A con qualcosa di più complesso.

Inizieremo con un display a cristalli liquidi o LCD (dall'inglese Liquid Crystal Display) dotato di 2 linee di 16 caratteri ciascuna. I display LCD più comuni reperibili in commercio, dispongono di una un'interfaccia ideata da Hitachi che nel tempo è diventata uno standard industriale utilizzato anche da altre case produttrici.

Questo tipo di interfaccia prevede che il display sia collegato al micro tramite un bus dati da 4 o 8 linee più 3 linee di controllo e le linee di alimentazione.

Diamo subito un'occhiata allo schema elettrico del circuito che andremo a realizzare per capire meglio le spiegazioni che seguiranno.

Lo schema elettrico del circuito è riportato di seguito:



Nella tabella seguente vengono descritte le funzioni di ogni singola linea disponibile per interfacciare il display. Le descrizioni riportate in grassetto indicano le linee effettivamente utilizzate dalla nostra applicazione d'esempio.

Pin	Nome	Funzione
1	GND	Ground Questo pin deve essere collegato con il negativo di alimentazione
2	VDD	Power supply. Questo pin deve essere colleato con i +5 volt di alimentazione
3	LCD	Liquid crystal driving voltage. A questo pin deve essere applicata una tensione variabile da 0 a 5 volt tramite un trimmer per regolare il contrasto del display
4	RS	Register select. Questo pin è una linea di controllo con cui si comunica al display se si sta inviando sul bus dati (linee da DB0 a DB7) un comando (RS = 0) oppure un dato (RS = 1)
5	R/W	Read, Write. Questo pin è un'altra linea di controllo con cui si comunica al display se si intende inviare un dato al display (R/W = 0) oppure leggere un dato dal display (R/W = 1).
6	E	Enable. Questo pin è un'altra linea di controllo con cui si può abilitare il display ad accettare dati ed istruzioni dal bus dati (E = 1).
7	DB0	Data bus line 0 - Su queste linee viaggiano i dati tra il micro ed il display LCD
8	DB1	Data bus line 1
9	DB2	Data bus line 2
10	DB3	Data bus line 3
11	DB4	Data bus line 4
12	DB5	Data bus line 5
13	DB6	Data bus line 6
14	DB7	Data bus line 7

Per ridurre al massimo i collegamenti tra il PIC ed il display LCD, in questa lezione utilizzeremo la modalità di collegamento dati a 4 bit utilizzando solo le linee DB4, DB5, DB6 e DB7. Le linee DB0, DB1, DB2 e DB3 non saranno utilizzate e collegate a massa.

Anche la linea R/W non verrà utilizzata e collegata direttamente a massa. In questo modo verrà selezionata la modalità di funzionamento in sola scrittura. In pratica potremo solo inviare dati all'LCD ma non riceverli.

Hello World !

Dopo aver realizzato il circuito d'esempio, compilare il sorgente riportato nel file [LCD1.ASM](#) e quindi programmate il PIC con il file .HEX ottenuto.

All'accensione del circuito, se tutto è andato per il meglio, apparirà sul display la seguente schermata.



ovvero "CIAO MONDO !" che è diventata ormai la frase di iniziazione che tutti gli aspiranti programmatori devono aver visualizzato almeno una volta in un loro programma. Non potevamo essere di certo noi a non rispettare questa tradizione. Il risultato non è molto esaltante, ma la sostanza di quello che siamo riusciti a fare è notevole e va studiata approfonditamente !

Se non riuscite a visualizzare nulla sul display, nonostante siate più che sicuri che il circuito sia stato realizzato correttamente e che il PIC sia stato ben programmato (ricordatevi a proposito di programmare l'oscillatore in modalità XT e di disabilitare il Watch Dog Timer), sarà forse necessario regolare il contrasto del display LCD agendo sul trimmer R2 connesso al pin 3 del display.

Le linee Enable (E) e Register Select (RS) dell'LCD

Per poter visualizzare una scritta sul display, il PIC deve inviare tutta una serie di comandi tramite le linee del bus dati (linee da DB4 a DB7). Per far questo utilizza due linee di controllo con cui comunica al display l'operazione di trasferimento che cerca di compiere sul bus.

Le due linee di controllo sono la linea Enable (pin 6 dell'LCD) e Register Select (pin 4 dell'LCD).

Con la linea Register Select (RS) il PIC segnala al display che il dato presente sul bus è un comando (RS=0) o un dato da visualizzare (RS=1). Tramite i comandi il PIC può segnalare al display il tipo di operazione da compiere, come ad esempio spostare il cursore o pulire lo schermo. Con i dati il PIC può inviare al display direttamente i caratteri ASCII da

visualizzare.

La linea Enable abilita il display a leggere il comando o il dato inviato sul bus dal PIC. Il PIC deve preoccuparsi di aver già inviato sul bus dati il comando o il dato giusto prima di mettere a 1 il segnale di enable.

Multiplex sul bus dati

Sia i comandi che i dati sono rappresentati da numeri a 8 bit, come è possibile inviarli al display se il bus dati è composto da sole 4 linee ?

Viene fatta in pratica una operazione detta di "multiplex", ovvero ogni byte viene scomposto in due gruppi di di 4 bit e quindi trasmessi sul bus dati in sequenza. Vengono inviati prima i quattro bit meno significativi seguiti dai quattro bit più significativi.

Nel nostro source di esempio, tutte le operazioni di trasmissione di dati e comandi verso il display vengono eseguite da una serie di subroutine presenti nel file LCD1.ASM semplificando così al massimo la complessità del nostro programma.

Prima di addentrarci nello studio delle singole subroutine vediamo come funziona il programma principale.

Analizziamo il sorgente LCD1.ASM

Nella prima parte del nostro source vengono definite alcune costanti:

```
;LCD Control lines

LCD_RS    equ 2 ;Register Select
LCD_E     equ 3 ;Enable

;LCD data line bus

LCD_DB4   equ 4 ;LCD data line DB4
LCD_DB5   equ 5 ;LCD data line DB5
LCD_DB6   equ 6 ;LCD data line DB6
LCD_DB7   equ 7 ;LCD data line DB7
```

Queste costanti definiscono l'associazione tra le linee del PIC (tutte connesse alla PORTA B) e le linee del display. Le singole definizioni verranno usate all'interno delle subroutine di gestione dell'LCD al posto dei singoli numeri di identificazione delle linee di I/O.

```
tmpLcdRegister res 2
msDelayCounter res 2
```

Di seguito viene allocato spazio per due registri: tmpLcdRegister , usato dalle routine di gestione dell'LCD e msDelayCounter usato dalla subroutine msDelay che genera dei ritardi software da 1 ms per il contenuto del registro W. Questa subroutine viene utilizzata sempre dalle subroutine di gestione dell'LCD per generare le temporizzazioni richieste durante la trasmissione di dati e comandi all'LCD.

Segue una parte di definizione delle linee di connessione tra il PIC ed il display e quindi arriva la prima chiamata a subroutine che ci interessa.

```
call    LcdInit
```

LcdInit è una subroutine che deve essere chiamata solo una volta all'inizio del programma e prima di qualsiasi altra subroutine di gestione dell'LCD. Essa si occupa di effettuare tutte le operazioni necessarie per inizializzare correttamente l'LCD e consentire, alle funzioni successive, di poter operare correttamente.

Con le istruzioni seguenti:

```
movlw    00H
call     LcdLocate
```

si posiziona il cursore del display sulla prima riga e prima colonna dello schermo. I caratteri inviati successivamente verranno visualizzati a partire da questa posizione. I quattro bit più significativi del valore caricato nel registro W con l'istruzione:

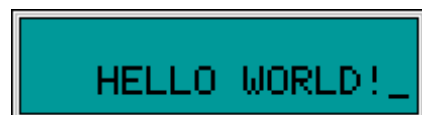
```
movlw    00H
```

contengono il numero della riga dove si vuole posizionare il cursore, i quattro bit meno significativi contengono il numero della colonna.

Provando a cambiare il valore nel registro W possiamo ottenere posizionamenti diversi. Con il valore 10H ad esempio otterremo la schermata:



con il valore 12H otterremo:



A questo punto per visualizzare ogni carattere della scritta vengono utilizzate le seguenti istruzioni:

```
movlw  'H'
call   LcdSendData
```

e così via per ogni lettere da visualizzare. L'incremento della posizione del cursore avviene automaticamente.

Subroutine di gestione del display LCD

Vediamo brevemente quali funzioni effettuano le subroutine di gestione del display LCD fornite con il sorgente LCD1.ASM.

Subroutine	Funzione
LcdInit	Questa subroutine si occupa di inizializzare il display LCD e di pulire lo schermo. Deve essere chiamata una sola volta e prima di qualsiasi altra subroutine di gestione dell'LCD. Non richiede alcun passaggio di parametri
LcdClear	Pulisce il contenuto dello schermo e riposiziona il cursore sulla prima colonna della prima riga. Non richiede alcun passaggio di parametri
LcdLocate	Serve a posizionare arbitrariamente il cursore all'interno dell'area visibile del display. Richiede il valore di riga e colonna per il posizionamento del cursore nel registro W. I bit da D0 a D3 contengono il valore di colonna (asse Y) mentre i bit da D4 a D7 il valore di riga (asse X). La numerazione delle righe parte da zero in alto. La numerazione delle colonne parte da 0 a sinistra.
LcdSendData	Serve ad inviare un carattere ASCII all'LCD da visualizzare nella posizione corrente del cursore. Richiede nel registro W il valore ASCII del carattere da visualizzare.
LcdSendCommand	Serve ad inviare un comando all'LCD. I comandi riconosciuti dall'LCD sono riportati sul datasheet dello stesso. Richiede nel registro W il valore ad 8 bit del comando da inviare.
LcdSendByte	Questa funzione viene utilizzata internamente dalle altre funzioni di gestione dell'LCD e di occupa effettuare lo shift di dati e comandi ad 8 bit sul bus dati a 4 bit.

Maggiori informazioni sull'uso dei display LCD possono essere ricavate direttamente dai datasheet che il fornitore dovrebbe poter essere in grado di fornirvi.


```

;*****
; Pic by example
; LCD1.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR    16F84
        RADIX        DEC
        INCLUDE      "P16F84.INC"

;Suppres MPASM warning message 302:
;"Register in operand not in bank 0.  Ensure that bank bits are correct"

        ERRORLEVEL   -302

;LCD Control lines

LCD_RS      equ      2      ;Register Select
LCD_E       equ      3      ;Enable

;LCD data line bus

LCD_DB4      equ      4      ;LCD data line DB4
LCD_DB5      equ      5      ;LCD data line DB5
LCD_DB6      equ      6      ;LCD data line DB6
LCD_DB7      equ      7      ;LCD data line DB7

        ORG          0CH

tmpLcdRegister res    2
msDelayCounter res    2

;Reset Vector

Start      ORG        00H

        bsf          STATUS,RP0      ;Swap to register bank 1

        movlw        00011111B      ;Set PORTA lines
        movwf        TRISA

        movlw        11111111B      ;Set PORTB lines
        movwf        TRISB

        bcf          PORTB,LCD_DB4   ;Set as output just the LCD's lines
        bcf          PORTB,LCD_DB5
        bcf          PORTB,LCD_DB6
        bcf          PORTB,LCD_DB7
        bcf          PORTB,LCD_E
        bcf          PORTB,LCD_RS

        bcf          STATUS,RP0      ;Swap to register bank 0

;LCD initialization

        call         LcdInit

;Locate LCD cursor on row 0, col 0

        movlw        10H
        call         LcdLocate

;Shows "HELLO WORLD" string on LCD

        movlw        'H'
        call         LcdSendData

```

```

        movlw    'E'
        call     LcdSendData

        movlw    'L'
        call     LcdSendData

        movlw    'L'
        call     LcdSendData

        movlw    'O'
        call     LcdSendData

        movlw    ' '
        call     LcdSendData

        movlw    'W'
        call     LcdSendData

        movlw    'O'
        call     LcdSendData

        movlw    'R'
        call     LcdSendData

        movlw    'L'
        call     LcdSendData

        movlw    'D'
        call     LcdSendData

        movlw    ' '
        call     LcdSendData

        movlw    '!'
        call     LcdSendData

```

```

foreverLoop

```

```

        goto     foreverLoop

```

```

;*****
;
; Delay subroutine
;
; W = Requested delay time in ms (clock = 4MHz)
;*****

```

```

msDelay

```

```

        movwf    msDelayCounter+1
        clrf     msDelayCounter+0

        ; 1 ms (about) internal loop

msDelayLoop
        nop
        decfsz   msDelayCounter+0,F
        goto     msDelayLoop
        nop

        decfsz   msDelayCounter+1,F
        goto     msDelayLoop

        return

```

```

;*****
;
; Init LCD
; This subroutine must be called before each other lcd subroutine
;*****

```

```

LcdInit

```

```

    movlw    30                ;Wait 30 ms
    call     msDelay

;*****
;
; Reset sequence
;*****

    bcf      PORTB,LCD_RS      ;Set LCD command mode

;Send a reset sequence to LCD

    bsf      PORTB,LCD_DB4
    bsf      PORTB,LCD_DB5
    bcf      PORTB,LCD_DB6
    bcf      PORTB,LCD_DB7

    bsf      PORTB,LCD_E       ;Enables LCD
    movlw    5                 ;Wait 5 ms
    call     msDelay
    bcf      PORTB,LCD_E       ;Disables LCD
    movlw    1                 ;Wait 1ms
    call     msDelay

    bsf      PORTB,LCD_E       ;Enables LCD
    movlw    1                 ;Wait 1ms
    call     msDelay
    bcf      PORTB,LCD_E       ;Disables LCD
    movlw    1                 ;Wait 1ms
    call     msDelay

    bsf      PORTB,LCD_E       ;Enables E
    movlw    1                 ;Wait 1ms
    call     msDelay
    bcf      PORTB,LCD_E       ;Disables E
    movlw    1                 ;Wait 1ms
    call     msDelay

    bcf      PORTB,LCD_DB4
    bsf      PORTB,LCD_DB5
    bcf      PORTB,LCD_DB6
    bcf      PORTB,LCD_DB7

    bsf      PORTB,LCD_E       ;Enables LCD
    movlw    1                 ;Wait 1ms
    call     msDelay
    bcf      PORTB,LCD_E       ;Disabled LCD
    movlw    1                 ;Wait 1ms
    call     msDelay

;Set 4 bit data bus length

    movlw    28H;
    call     LcdSendCommand

;Entry mode set, increment, no shift

    movlw    06H;
    call     LcdSendCommand

;Display ON, Curson OFF, Blink OFF

    movlw    0CH
    call     LcdSendCommand

;Clear display

    call     LcdClear

```

```
return
```

```
; *****
;
; Clear LCD
; *****
```

```
LcdClear
```

```
    ;Clear display

    movlw    01H
    call     LcdSendCommand

    movlw    2                ;Wait 2 ms
    call     msDelay

    ;DD RAM address set 1st digit

    movlw    80H;
    call     LcdSendCommand

    return
```

```
; *****
;
; Locate cursor on LCD
; W = D7-D4 row, D3-D0 col
; *****
```

```
LcdLocate
```

```
    movwf    tmpLcdRegister+0

    movlw    80H
    movwf    tmpLcdRegister+1

    movf     tmpLcdRegister+0,W
    andlw    0FH
    iorwf    tmpLcdRegister+1,F

    btfsc    tmpLcdRegister+0,4
    bsf      tmpLcdRegister+1,6

    movf     tmpLcdRegister+1,W
    call     LcdSendCommand

    return
```

```
; *****
;
; Send a data to LCD
; *****
```

```
LcdSendData
```

```
    bsf      PORTB, LCD_RS
    call     LcdSendByte
    return
```

```
; *****
;
; Send a command to LCD
; *****
```

```
LcdSendCommand
```

```
    bcf      PORTB, LCD_RS
    call     LcdSendByte
    return
```

```
; *****
;
; Send a byte to LCD by 4 bit data bus
```

```
;*****
```

```
LcdSendByte
```

```

;Save value to send

movwf    tmpLcdRegister

;Send higher four bits

bcf      PORTB, LCD_DB4
bcf      PORTB, LCD_DB5
bcf      PORTB, LCD_DB6
bcf      PORTB, LCD_DB7

btfsc    tmpLcdRegister, 4
bsf      PORTB, LCD_DB4
btfsc    tmpLcdRegister, 5
bsf      PORTB, LCD_DB5
btfsc    tmpLcdRegister, 6
bsf      PORTB, LCD_DB6
btfsc    tmpLcdRegister, 7
bsf      PORTB, LCD_DB7

bsf      PORTB, LCD_E      ;Enables LCD
movlw    1                 ;Wait 1ms
call     msDelay
bcf      PORTB, LCD_E      ;Disabled LCD
movlw    1                 ;Wait 1ms
call     msDelay

;Send lower four bits

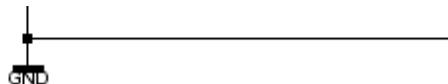
bcf      PORTB, LCD_DB4
bcf      PORTB, LCD_DB5
bcf      PORTB, LCD_DB6
bcf      PORTB, LCD_DB7

btfsc    tmpLcdRegister, 0
bsf      PORTB, LCD_DB4
btfsc    tmpLcdRegister, 1
bsf      PORTB, LCD_DB5
btfsc    tmpLcdRegister, 2
bsf      PORTB, LCD_DB6
btfsc    tmpLcdRegister, 3
bsf      PORTB, LCD_DB7

bsf      PORTB, LCD_E      ;Enables LCD
movlw    1                 ;Wait 1ms
call     msDelay
bcf      PORTB, LCD_E      ;Disabled LCD
movlw    1                 ;Wait 1ms
call     msDelay

return

END
```

Come potete vedere la base del circuito che andremo a realizzare è identica a quella del passo precedente con la sola aggiunta del circuito integrato U3, del connettore a vaschetta DB9 per il collegamento alla porta seriale del PC ed una manciata di componenti accessori.

L'integrato U3, un MAX232 prodotto dalla Maxim (vedi datasheet su <http://www.maxim-ic.com>), si occupa di convertire i segnali RS232 dai +/-12 volt necessari per trasmettere e ricevere dati sulla porta seriale ai 0/5 volt TTL gestibili direttamente dalle porte del PIC.

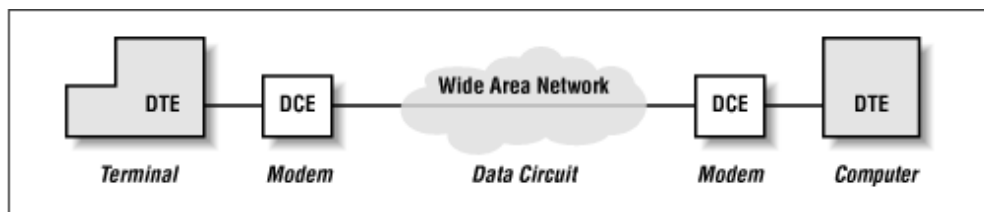
Ma vediamo in dettaglio come funziona la comunicazione seriale in RS232.

Cos'è e a cosa serve l'RS232

Lo standard RS232 definisce una serie di specifiche per la trasmissione seriale di dati tra due dispositivi denominati DTE (Data Terminal Equipment) e DCE (Data Communication Equipment). Come si può vagamente intuire dal nome, il Data Communication Equipment è un dispositivo che si occupa di gestire una comunicazione dati mentre il Data Terminal Equipment è un dispositivo che si occupa di generare o ricevere dati.

In pratica l'RS232 è stata creata per connettere tra loro un terminale dati (nel nostro caso un computer) con un modem per la trasmissione a distanza dei dati generati.

Per avere una connessione tra due computer è quindi necessario disporre di quattro dispositivi come visibile in figura: un computer (DTE) collegato al suo modem (DCE) ed un altro modem (DCE) collegato al suo computer (DTE). In questo modo qualsiasi dato generato dal primo computer e trasmesso tramite RS232 al relativo modem verrà trasmesso da questo al modem remoto che a sua volta provvederà ad inviarlo al suo computer tramite RS232. Lo stesso vale per il percorso a ritroso.



Per usare la RS232 per collegare tra loro due computer vicini senza interporre tra loro alcun modem dobbiamo simulare in qualche modo le connessioni intermedie realizzando un cavo NULL MODEM o cavo invertente, ovvero un cavo in grado di far scambiare direttamente tra loro i segnali dai due DTE come se tra loro ci fossero effettivamente i DCE.

Per connettere il PC al nostro circuito simuleremo invece direttamente un DCE facendo credere al PC di essere collegato ad un modem. Prima di fare questo diamo uno sguardo in dettaglio al principio di funzionamento di una comunicazione seriale.

La comunicazione seriale asincrona

Per consentire la trasmissione di dati tra il PC ed il modem, lo standard RS232 definisce una serie di specifiche elettriche e meccaniche. Una di queste riguarda il tipo di comunicazione seriale che si vuole implementare che può essere sincrona o asincrona.

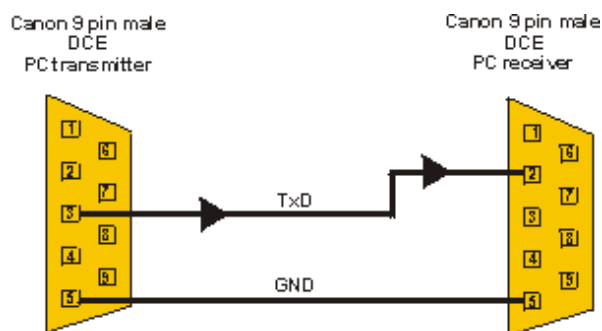
Nel nostro caso analizzeremo solo la comunicazione seriale asincrona ignorando completamente quella sincrona in quanto più complessa e non disponibile sui normali PC.

Una comunicazione seriale consiste in genere nella trasmissione e ricezione di dati da un punto ad un altro usando una sola linea elettrica. In pratica se desideriamo trasmettere un intero byte dobbiamo prendere ogni singolo bit in esso contenuto ed inviarlo in sequenza sulla stessa linea elettrica, un po' come avviene per la trasmissione in codice morse. La differenza sostanziale sta nel fatto che a generare e ricevere dati non c'è il telegrafista ma un computer per cui le velocità di trasmissione raggiungibili sono molto superiori.

Facciamo subito un esempio pratico e vediamo come fa un PC a trasmettere, ad esempio, il carattere A usando la RS232.

Non è necessario ovviamente realizzare gli esempi riportati di seguito in quanto presuppongono l'uso di una coppia di PC ed un oscilloscopio non sempre disponibili nei nostri mini-laboratori da hobbysta. Per comprendere il funzionamento di quanto esposto è sufficiente fare riferimento alle figure a corredo.

Se prendiamo una coppia di fili e colleghiamo tra loro le porte seriali di due PC (che denomineremo PC trasmittente e PC ricevente) secondo lo schema riportato in figura:



otterremo la più semplice delle connessioni in RS232.

La linea Transmit Data (TXD) presente sul pin 3 del connettore DB9 maschio di cui il vostro PC è dotato è connessa alla linea Receive Data (RXD) presente sul pin 2 del secondo PC. Le masse (GND) presenti sul pin 5 di entrambe i PC sono connesse tra loro.

Per osservare i segnali generati dal PC trasmittente durante la trasmissione seriale colleghiamo tra la linea TXD e la linea GND un oscilloscopio e lanciamo in esecuzione su entrambe i PC un programma di emulazione terminale (tipo Hyperterminal o simili).

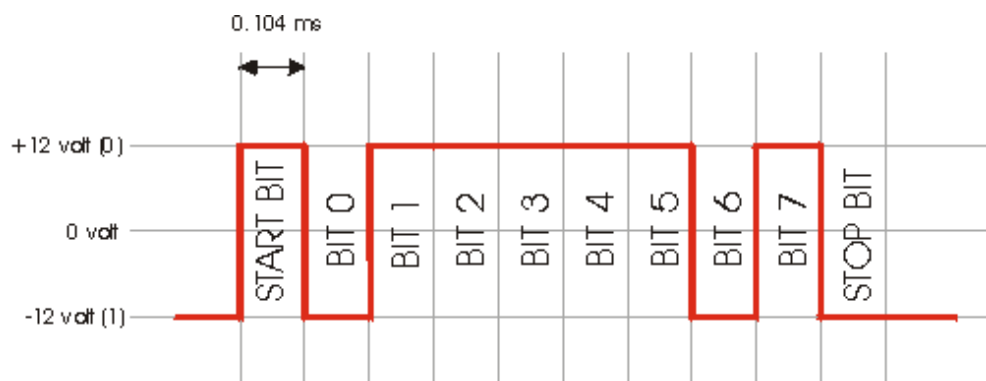
Configuriamo le porte seriali di entrambe i PC a 9600 baud, 8 data bit, 1 stop bit, no parity e disabilitiamo il controllo di flusso (handshake) sia hardware che xon/xoff. In questo stato qualsiasi cosa digiteremo sul PC trasmittente verrà inviata immediatamente sulla porta seriale. Assicuriamoci inoltre che il programma di emulazione terminale scelto sia opportunamente configurato per usare la porta seriale su cui siamo connessi (COM1 o COM2).

Proviamo a digitare la lettera A maiuscola e verifichiamo se è stata correttamente ricevuta sul PC ricevente. Fatto questo controllo andiamo a vedere sull'oscilloscopio che tipo di segnali sono stati generati per effettuare la trasmissione.

Quando non c'è nessuna trasmissione in corso la tensione sulla linea TXD è di -12 volt corrispondente alla condizione logica 1. Per indicare al PC ricevente che la trasmissione ha inizio, il PC trasmittente porta a +12 volt la linea TXD per un tempo pari all'inverso della frequenza di trasmissione ovvero al tempo di trasmissione di un singolo bit.

Nel nostro caso, avendo scelto di trasmettere a 9600 bit per secondo, la tensione di alimentazione rimarrà a +12 volt per: $1/9600 = 0.104$ mS.

Questo segnale viene denominato START BIT ed è sempre presente all'inizio di trasmissione di ogni singolo byte. Dopo lo start bit vengono trasmessi in sequenza gli otto bit componenti il codice ASCII del carattere trasmesso partendo dal bit meno significativo. Nel nostro caso la lettera A maiuscola corrisponde al valore binario 01000001 per cui la sequenza di trasmissione sarà la seguente:



Una volta trasmesso l'ottavo bit (bit 7), il PC aggiunge automaticamente un ultimo bit a 1 denominato STOP BIT ad indicare l'avvenuta trasmissione dell'intero byte. La stessa sequenza viene ripetuta per ogni byte trasmesso sulla linea.

Aggiungendo al nostro cavo seriale una connessione tra il pin TXD (pin 3) del PC ricevente con il pin RXD (pin 2) del PC trasmittente, potremo effettuare una trasmissione RS232 bidirezionale. Il cavo che abbiamo ottenuto è il più semplice cavo NULL MODEM in grado di mettere in collegamento tra loro due DTE.

Come collegare il nostro circuito d'esempio

Come accennato prima il nostro circuito d'esempio simula un dispositivo DCE. Questo significa che il cavo che dovremo realizzare non dovrà essere di tipo NULL MODEM o INVERTENTE ma DRITTO ovvero con i pin numerati allo stesso modo connessi tra loro. Questo tipo di cavo è identico a quelli che vengono usati per connettere al PC un modem esterno.

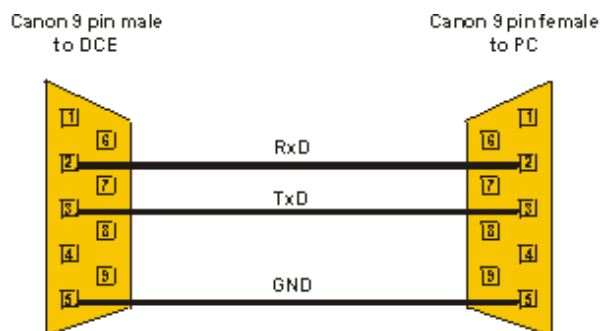
Dato che i dispositivi DTE sono sempre dotati di connettore DB9 maschio, il nostro circuito, essendo un DCE, avrà un connettore DB9 femmina la cui piedinature è riportata nella seguente tabella:

Pin 1	Received Line Signal Detector (Data Carrier Detect)
Pin 2	Received Data
Pin 3	Transmit Data
Pin 4	Data Terminal Ready
Pin 5	Signal Ground
Pin 6	Data Set Ready
Pin 7	Request To Send
Pin 8	Clear To Send
Pin 9	Ring Indicator

In alcuni casi i PC sono dotati di vecchi connettori DB25 anziché DB9 per cui occorre far riferimento alla seguente tabella per ricavare le corrette equivalenze:

Pin 1	Protective Ground
Pin 2	Transmit Data
Pin 3	Received Data
Pin 4	Request To Send
Pin 5	Clear To Send
Pin 6	Data Set Ready
Pin 7	Signal Ground
Pin 8	Received Line Signal Detector (Data Carrier Detect)
Pin 20	Data Terminal Ready
Pin 22	Ring Indicator

Il cavo di collegamento tra il PC ed il nostro circuito dovrà essere intestato a sua volta con un connettore femmina da un lato per poter essere inserito nella seriale del PC ed un connettore maschio dall'altro per poter essere inserito nel connettore del nostro circuito di prova. I collegamenti interni al cavo da usare sono riportati nella seguente figura.



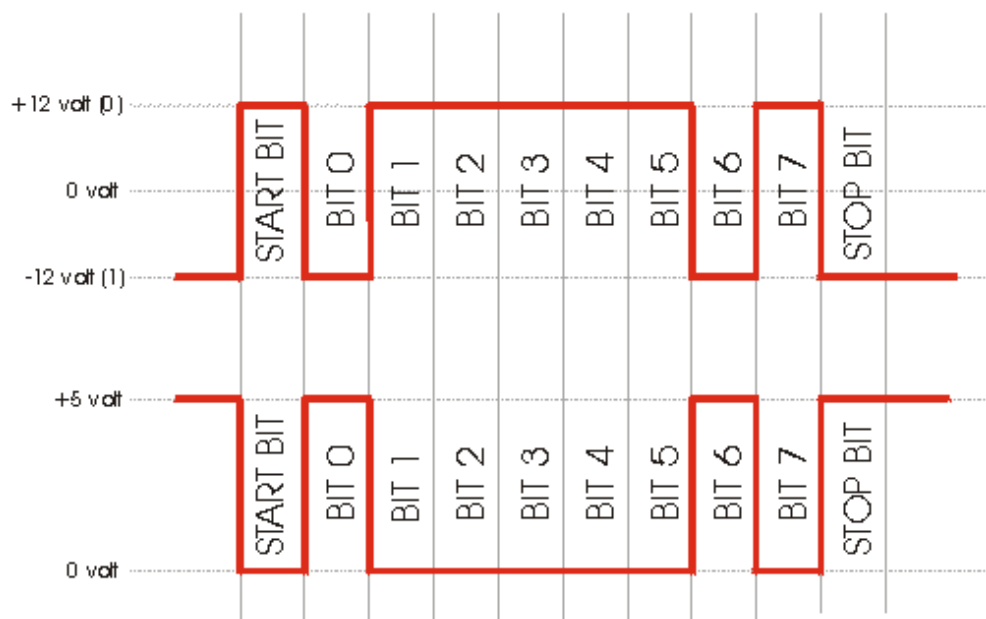
Funzionamento del MAX232

Come accennato prima, nel nostro circuito d'esempio useremo un driver RS232, ovvero un integrato in grado di convertire i segnali a +/- 12 volt tipici della RS232 in segnali a 0/5 volt gestibili dalle porte del PIC.

Seguendo lo schema elettrico del nostro circuito d'esempio vediamo che il segnale di trasmissione proveniente dal PC entra dal pin 3 del connettore DB9 femmina sul pin 13 di U3. Sul pin 12 di U3 sarà presente un segnale a 0 volt quando sul pin 13

ci saranno +12 volt e 5 volt quando sul pin 13 ci saranno -12 volt. Il segnale presente sul pin 12 di U3 viene quindi inviato alla linea RA1 della porta A del PIC che in questo caso farà da linea di ricezione.

Sul pin 18 del PIC (RA1) avremo quindi la seguente corrispondenza di segnali con la linea TXD del PC.



Viceversa sul pin 17 (RA0) il PIC genera i segnali da inviare al PC a livello TTL che vengono convertiti in segnali RS232 da U3 tramite i pin 11 (ingresso TTL) e 14 (uscita RS232) e quindi inviati al PC tramite il pin 2 del connettore J2.

Applicazione d'esempio

Mettiamo finalmente mano al source della nostra applicazione d'esempio e vediamo come ricevere e trasmettere dati dal nostro PIC.

Nel file [LCDTERM.ASM](#) troverete il source completo del nostro terminale d'esempio.

Una volta montato il nostro circuito d'esempio e programmato correttamente il PIC16F84 possiamo collegare al nostro PC il circuito e fornire alimentazione. Sul display apparirà il cursore lampeggiante in alto a sinistra.

A questo punto lanciamo in esecuzione un programma qualsiasi di emulazione terminale e configuriamolo per usare la porta seriale a cui è collegato il circuito a 9600 baud, 8 data bit, 1 stop bit e no parity. Assicuriamoci inoltre che non sia abilitato alcun controllo di flusso dei dati sulla seriale sia esso hardware che xon/xoff.

Proviamo ora a premere qualche tasto sulla tastiera del PC ed osserviamo come i caratteri digitati vengano visualizzati anche sul display LCD del nostro circuito. Premendo i tasti CTRL-L potremo pulire lo schermo dell'LCD e visualizzare nuove scritte.

Analizziamo il sorgente

Andiamo ad analizzare ora il sorgente LCDTERM.ASM del firmware della nostra applicazione d'esempio.

Partiamo dalla linea 24 dove troviamo le seguenti direttive:

```
TX equ 0 ;Tx data
RX equ 1 ;Rx data
```

in cui vengono assegnate alle costanti TX e RX rispettivamente le linee di trasmissione (TX) e ricezione (RX) del PIC. In questa applicazione in realtà non viene ancora usata la linea di trasmissione in quanto il nostro miniterminale è in grado per ora solo di ricevere caratteri.

Queste due costanti vengono utilizzate rispettivamente dalle subroutine di trasmissione e ricezione di caratteri via RS232: TxChar (vedi linea 421) ed RxChar (vedi linea 483). Queste due subroutine consentono in pratica di trasmettere e ricevere byte in modalità seriale asincrona a 9600 bps, 8 bit dati, 1 stop bit e nessuna parità.

Per trasmettere un carattere sulla linea TX basta inserire nel registro W il valore da trasmettere ed effettuare una chiamata alla subroutine TxChar. Ipotizzando di voler trasmettere il carattere 'A' al PC dovremo inserire il seguente codice:

```
movlw 'A'
call TxChar
```

Per ricevere caratteri l'operazione è leggermente più complessa. Prendiamo in esame il nostro esempio a partire dalla linea 129:

```
MainLoop
    btfsc PORTA,RX
    goto MainLoop

    call RxChar
```

In pratica il nostro programma esegue un loop infinito finchè non rileva uno stato logico 0 sulla linea RX. Quando questo avviene significa che molto probabilmente è arrivato lo START BIT dal PC e che, secondo quanto detto sopra, arriveranno in sequenza i bit appartenenti al dato trasmesso dal PC.

In questo caso viene chiamata la RxChar che si occuperà di leggere ogni singolo bit ricevuto, compattarli in un unico byte e restituire il valore del byte così ricevuto nel registro ShiftReg.

Una volta lanciata la RxChar azzerava il registro ShiftReg in cui verranno memorizzati i bit man mano che vengono ricevuti

```
RxChar
    clrf ShiftReg
```

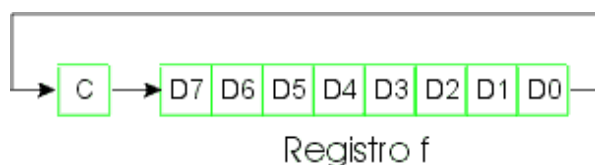
quindi mette a 8 il registro BitCount usato per il conteggio del numero di bit in arrivo

```
movlw 8
movwf BitCount
```

a questo punto attende un periodo pari a circa 1 bit e mezzo in modo da far scorrere il tempo necessario alla trasmissione dello start bit e campionare il valore del BIT 0 circa a metà del tempo di durata.

```
DELAY BIT_DELAY+BIT_DELAY/2 ;Wait 1.5 bit
```

a questo punto legge lo stato della linea RX ed inserisce il valore letto nel flag di CARRY (C) del registro STATUS e quindi effettua una istruzione di ROTATE RIGHT F TROUGHOUT CARRY (RRF) con il registro ShiftReg in modo da spostare verso destra tutti i bit del registro ShiftReg ed inserire nel bit più significativo il valore appena letto dalla linea RX come riportato nella seguente figura:



questa lettura avviene per otto volte ad intervalli di tempo pari alla durata di un bit in modo da campionare il valore della linea RX sempre al centro del bit in ricezione.

```
wDB
    btfss PORTA,RX
    goto RxBitL

RxBitH
    nop
    bsf STATUS,C
    goto RxShift

RxBitL
    bcf STATUS,C
    goto RxShift

RxShift
    nop
    rrf ShiftReg,F
```

attende per un periodo di tempo pari ad 1 bit

```
DELAY BIT_DELAY
```

continua a campionare se non ha ancora letto tutti ed otto i bit

```

    decfsz BitCount,F
    goto wDB

```

esce da RxChar dopo aver letto l'ultimo bit

```

    return

```

a questo punto nel registri ShiftReg dovrebbe esserci il byte trasmesso dal PC

Una volta letto il byte proveniente dal PC il nostro programma d'esempio controlla se il byte ricevuto è un carattere di controllo oppure un normale carattere da visualizzare su LCD.

L'unico carattere di controllo implementato dal nostro miniterminale è il Form Feed (FF) corrispondente al codice ASCII decimale 12. La trasmissione di questo carattere verso una stampante determina l'avanzamento di un foglio di carta. Nel nostro caso pulisce il contenuto dell'LCD. Il form feed può essere trasmesso dal nostro simulatore di terminale su PC premendo il tasto CTRL seguito dalla lettera L.

Questa è la parte di codice che gestisce la ricezione di un Form Feed:

```

CheckFormFeed

```

```

    movlw 12
    xorwf ShiftReg,W
    btfss STATUS,Z
    goto _CheckFormFeed

    clrf xCurPos
    clrf yCurPos
    call LcdClear
    goto MainLoop

```

```

_CheckFormFeed

```

in pratica viene controllato se il valore ricevuto dalla subroutine RxChar è pari a 12. In caso affermativo vengono azzerati i registri xCurPos e yCurPos che mantengono il valore X,Y del cursore carattere su display. Quindi viene chiamata la subroutine LcdClear che si occupa di inviare i comandi corretti al display LCD per azzerarne il contenuto.

Nel caso non sia stato trasmetto un FF dal PC, il carattere ricevuto viene inviato nudo e crudo al display con il seguente codice:

```

    movf ShiftReg,W
    call putchar

```

e quindi si ritorna ad attendere lo START BIT del prossimo carattere con la seguente istruzione:

```

    goto MainLoop

```

La subroutine putchar in pratica invia il valore contenuto nel registro W al display LCD nella posizione in cui si trova il cursore carattere (xCurPos e yCurPos), quindi si occupa di mandare a capo il cursore se si è raggiunto il fine riga o di riportarlo alla prima riga se si è raggiunto il fine display, In tutti i casi i registri xCurPos ed yCurPos vengono aggiornati alla prossima posizione in cui poter scrivere il successivo carattere ricevuto dal PC.

LCDPRINT un programma d'esempio per l'utilizzo del nostro miniterminale



[lcdprint.zip](#)

Da questo link potrete scaricare un semplice programma d'esempio per l'uso del nostro miniterminale RS232. Il programma di chiama LCDPRINT e funziona in ambiente MS/DOS o prompt MS/DOS sotto Windows 95/98/ME. Non è assolutamente in grado di funzionare sotto Windows NT/2000/XP.

LCDPRINT permette di visualizzare messaggi sul nostro miniterminali nel modo più semplice. Basta digitare dal prompt di MS/DOS il comando LCDPRINT seguito dal numero di porta seriale a cui è connesso il nostro miniterminale e la stringa da visualizzare tra doppi apici.

Se vogliamo visualizzare ad esempio la scritta "Ciao a tutti" sul miniterminale connesso alla porta COM2 dovremo digitare:

LCDPRINT /COM2 "Ciao a tutti !"

Le applicazioni possibili per questo semplice programma sono molte. Lo potremmo usare ad esempio per visualizzare dei messaggi durante boot di Windows 95 inserendo il comando all'interno del file AUTOEXEC.BAT.

Spunti per un pò di esercitazioni

La nostra applicazione d'esempio è stata sviluppata nel modo più semplice possibile per permettere a chiunque di capirne il funzionamento senza perdersi in centinaia di linee di codice. Questo comporta ovviamente una serie di limitazioni nelle funzionalità che questo miniterminale è in grado di dare.

Chi volesse fare un pò di esercizio può tentare di ampliare il numero di caratteri di controllo riconosciuto dal miniterminale quali ad esempio il Carriage Return, il New Line o il Backspace.

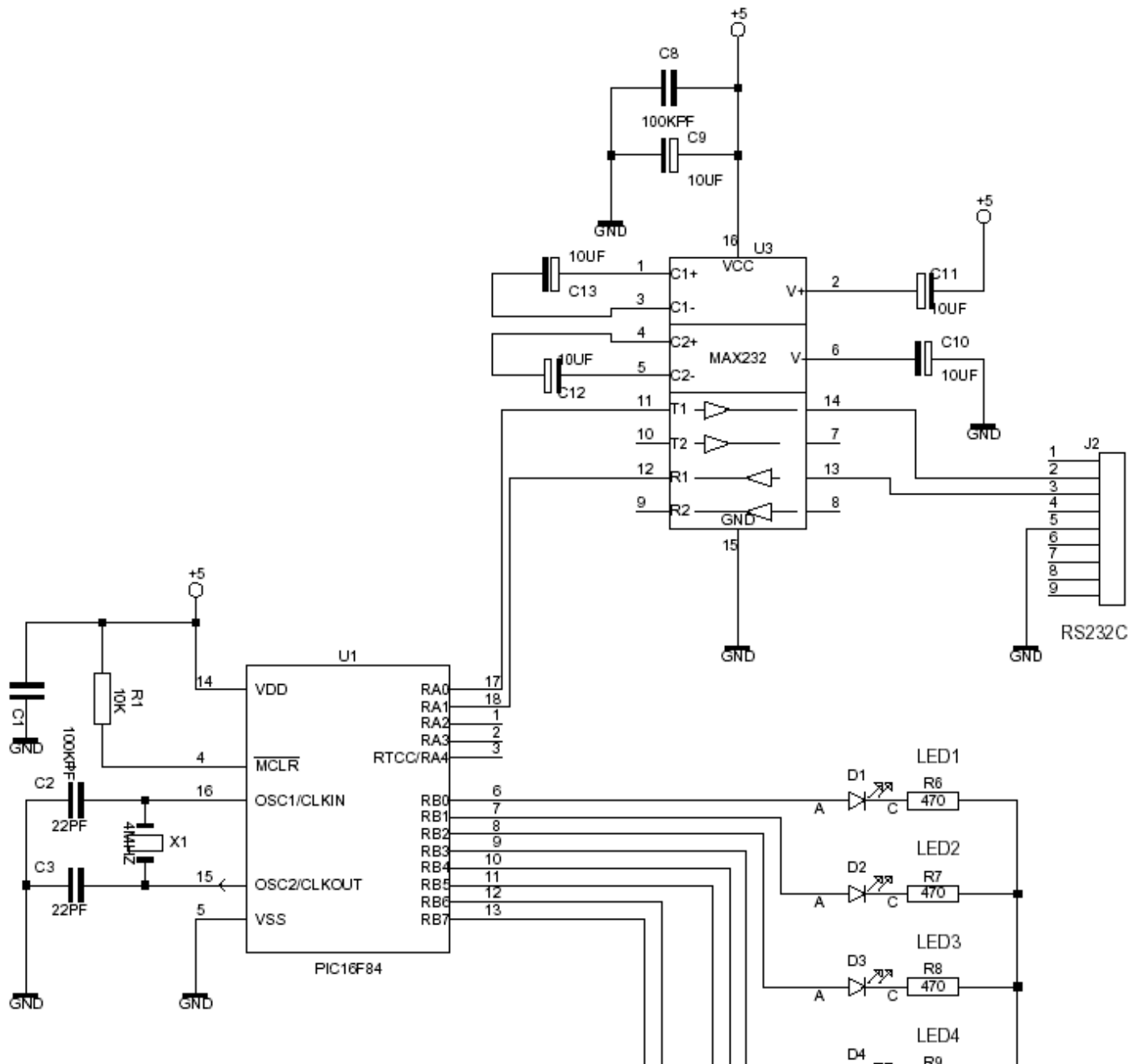
Chiunque abbia voglia di realizzare questi esercizi può inviare direttamente all'autore all'indirizzo sergio@tanzilli.com il proprio lavoro per una eventuale pubblicazione tra gli esempi del corso.

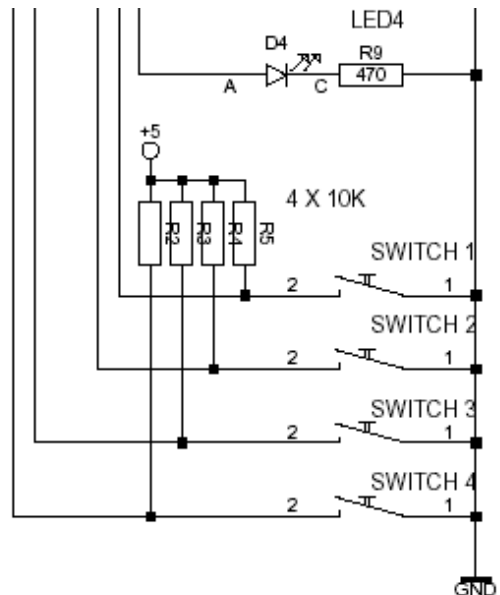
Un altro esempio con l'interfaccia RS232

Vediamo ora un altro esempio di utilizzo della RS232. In questo caso realizzeremo un circuito che sfrutta anche le capacità trasmissive messe a disposizione dalla subroutine TxChar.

Si tratta in pratica di una evoluzione del circuito già presentato in una delle lezioni precedenti con 4 led e 4 switch a cui abbiamo aggiunto ora una interfaccia RS232 per poter comandare i quattro led e leggere da PC lo stato dei quattro switch.

Nella figura seguente viene riportato lo schema elettrico completo:





Dai seguenti link potete scaricare il sorgente del firmware per il PICmicro ed il semplice programma per MS/DOS RS232IO con cui è possibile inviare dal PC i comandi di accensione dei singoli led e leggere lo stato corrente dei quattro switch.



[rs232io.asm](#)



[rs232io.zip](#)

Protocollo di comunicazione con il PC

Come già detto il nostro circuito è dotato di quattro LED denominati LED1, LED2, LED3 e LED4 e quattro pulsanti denominato SWITCH1, SWITCH2, SWITCH3 e SWITCH4.

Tramite un semplice protocollo possiamo decidere quale dei quattro led accendere oppure leggere lo stato di uno qualsiasi dei quattro pulsanti.

Il protocollo consiste in una serie di codici di controllo che il PC può trasmettere al nostro circuito tramite la seriale. La velocità di trasferimento è la solita 9600 baud, 8 data bit, 1 stop bit, no parity.

Comandi di accensione LED

Possiamo accendere ogni singolo LED inviando dal PC i seguenti comandi:

00h	Accensione LED 1
01h	Accensione LED 2
02h	Accensione LED 3
03h	Accensione LED4

Comandi di spegnimento LED

Possiamo spegnere ogni singolo LED inviando dal PC i seguenti comandi:

10h	Spegnimento LED 1
11h	Spegnimento LED 2
12h	Spegnimento LED 3
13h	Spegnimento LED4

Lettura stato pulsanti

Per leggere lo stato di tutti e quattro i pulsanti basta inviare un unico comando:

20h	Lettura stato pulsanti
------------	------------------------

Quando il PIC riceve questo comando dalla RS232 legge lo stato dei bit RB4, RB5, RB6 ed RB7 ed invia un unico codice al PC che riflette lo stato dei quattro pulsanti. Di questo codice solo i bit 0,1,2,3 indicano lo stato dei pulsanti secondo la seguente tabella.

BIT 0	0 = SWITCH 1 rilasciato, 1 = SWITCH 1 premuto
BIT 1	0 = SWITCH 2 rilasciato, 1 = SWITCH 2 premuto
BIT 2	0 = SWITCH 3 rilasciato, 1 = SWITCH 3 premuto
BIT 3	0 = SWITCH 4 rilasciato, 1 = SWITCH 4 premuto

Per cui se ad esempio solo lo SWITCH 1 risulta premuto il codice di risposta sarà 01h (00000001 binario). Se risultano premuti sia lo SWITCH 2 che il 4 il codice di risposta 0Ah (00001010 binario).

Programma di prova

Il programma di prova RS232IO.EXE consente di provare il circuito immediatamente. Il programma funziona in ambiente MS/DOS o prompt MS/DOS sotto Win 95/98.

Ipotizzando di aver collegato alla COM2 il nostro circuito, per accendere il LED 1 sarà sufficiente digitare:

```
RS232IO /COM2 /LED1=ON
```

Se ora vogliamo spegnere il LED 1 ed accendere il LED 4 potremo digitare:

```
RS232IO /COM2 /LED1=OFF /LED4=ON
```

Se ora vogliamo accendere solo il LED 3 senza modificare lo stato degli altri led:

```
RS232IO /COM2 /LED3=ON
```

Possiamo anche richiedere lo stato corrente dei singoli switch con un unico comando:

```
RS232IO /COM2 /SWITCH
```

Il programma risponderà con qualcosa del genere:

```
Switch 1: off
Switch 2: off
Switch 3: ---> Active
Switch 4: off
```

Ad indicare che solo lo SWITCH 3 risulta premuto.

```

;*****
; Pic by example
; LCDTERM.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"

;Suppress the following MPASM warning message [# 302]:
;"Register in operand not in bank 0.  Ensure that bank bits are correct"

        ERRORLEVEL     -302

;Flag configuration

        __CONFIG        3FF1H

;RS232 lines

TX      equ      0      ;Tx data
RX      equ      1      ;Rx data

;LCD Control lines

LCD_RS  equ      2      ;Register Select
LCD_E   equ      3      ;Enable

;LCD data line bus

LCD_DB4 equ      4      ;LCD data line DB4
LCD_DB5 equ      5      ;LCD data line DB5
LCD_DB6 equ      6      ;LCD data line DB6
LCD_DB7 equ      7      ;LCD data line DB7

;*****
; Clock frequency related constant (4 MHz)
;*****

BIT_DELAY equ      23      ;Bit delay a 9600 bps

;*****
; MACRO - Delay subroutine with watch dog timer clearing
;
; Macro parameters:
;
;   VALUE:      Delay obtained = ((VALUE-1)*4+5)*(1/(Fosc/4))
;
;*****

DELAY      MACRO      VALUE
            LOCAL      REDO

            movlw      VALUE
            movwf      TmpRegister

REDO

            clrwdt                      ;Clear watch dog timer

            decfsz     TmpRegister,F
            goto       REDO

        ENDM

;*****
;

```

```

; FILE REGISTER
;*****

                ORG      0CH

;Register used by LCD subroutines

tmpLcdRegister  res      2

;Register used by msDelay subroutine and DELAY macro

msDelayCounter  res      2
TmpRegister     res      1

;Register used by RS232 subroutines

ShiftReg        res      1      ;Shift register
BitCount        res      1      ;Bit counter

;Cursor location

xCurPos         res      1
yCurPos         res      1
putTempReg      res      1

                ;Reset Vector
;*****
; RESET VECTOR
;*****

Start           ORG      00H

                bsf      STATUS,RP0      ;Swap to register bank 1

                movlw    00011111B      ;Sets the whole PORTA as input
                movwf    TRISA

                movlw    11111111B      ;Sets the whole PORTB as input
                movwf    TRISB

                bcf      PORTA, TX      ;Sets TX line as output
                bcf      PORTB, LCD_DB4 ;Sets LCD data and control lines as output
                bcf      PORTB, LCD_DB5
                bcf      PORTB, LCD_DB6
                bcf      PORTB, LCD_DB7
                bcf      PORTB, LCD_E
                bcf      PORTB, LCD_RS

                bcf      STATUS,RP0      ;Swap to register bank 0

                ;LCD initialization

                call     LcdInit

                ;Put terminal cursor on 0,0 position

                clrf     xCurPos
                clrf     yCurPos

                ;Wait until receives a start bit from RS232 line

MainLoop

                btfsc    PORTA, RX      ;Received a start bit ?
                goto     MainLoop      ;No, wait.

                call     RxChar          ;Yes, read the byte on receiving...

CheckFormFeed

                movlw    12

```

```

    xorwf    ShiftReg,W
    btfss    STATUS,Z
    goto     _CheckFormFeed

```

```

    clrf     xCurPos
    clrf     yCurPos
    call     LcdClear
    goto     MainLoop

```

_CheckFormFeed

```

    movf     ShiftReg,W
    call     putchar

    goto     MainLoop

```

```

; *****
;
; Delay subroutine
;
; W = Requested delay time in ms (clock = 4MHz)
; *****

```

msDelay

```

    movwf    msDelayCounter+1
    clrf     msDelayCounter+0

    ; 1 ms (about) internal loop

```

msDelayLoop

```

    nop
    decfsz   msDelayCounter+0,F
    goto     msDelayLoop
    nop

    decfsz   msDelayCounter+1,F
    goto     msDelayLoop

    return

```

```

; *****
;
; Put a char to xCurPos, yCurPos position on LCD
;
; W = Char to show
; xCurPos = x position
; yCurPos = y position
;
; xCurPos and yCurPos will be increase automatically
; *****

```

putchar

```

    movwf    putTempReg

    swapf    yCurPos,W
    iorwf    xCurPos,W
    call     LcdLocate

    movf     putTempReg,W
    call     LcdSendData

    incf     xCurPos,F
    movlw    16
    xorwf    xCurPos,W
    btfss    STATUS,Z
    goto     moveLcdCursor

    clrf     xCurPos

    incf     yCurPos,F
    movlw    2

```

```

xorwf    yCurPos,W
btfss    STATUS,Z
goto     moveLcdCursor

```

```

clrf     yCurPos

```

```

moveLcdCursor

```

```

swapf    yCurPos,W
iorwf    xCurPos,W
call     LcdLocate

```

```

return

```

```

;*****
;
; Init LCD
; This subroutine must be called before each other lcd subroutine
;*****
;

```

```

LcdInit

```

```

movlw    30                ;Wait 30 ms
call     msDelay

```

```

;*****
;
; Reset sequence
;*****
;

```

```

bcf       PORTB,LCD_RS     ;Set LCD command mode

```

```

;Send a reset sequence to LCD

```

```

bsf       PORTB,LCD_DB4
bsf       PORTB,LCD_DB5
bcf       PORTB,LCD_DB6
bcf       PORTB,LCD_DB7

```

```

bsf       PORTB,LCD_E      ;Enables LCD
movlw     5                ;Wait 5 ms
call      msDelay
bcf       PORTB,LCD_E      ;Disables LCD
movlw     1                ;Wait 1ms
call      msDelay

```

```

bsf       PORTB,LCD_E      ;Enables LCD
movlw     1                ;Wait 1ms
call      msDelay
bcf       PORTB,LCD_E      ;Disables LCD
movlw     1                ;Wait 1ms
call      msDelay

```

```

bsf       PORTB,LCD_E      ;Enables E
movlw     1                ;Wait 1ms
call      msDelay
bcf       PORTB,LCD_E      ;Disables E
movlw     1                ;Wait 1ms
call      msDelay

```

```

bcf       PORTB,LCD_DB4
bsf       PORTB,LCD_DB5
bcf       PORTB,LCD_DB6
bcf       PORTB,LCD_DB7

```

```

bsf       PORTB,LCD_E      ;Enables LCD
movlw     1                ;Wait 1ms
call      msDelay
bcf       PORTB,LCD_E      ;Disabled LCD
movlw     1                ;Wait 1ms
call      msDelay

```

```

;Set 4 bit data bus length

movlw    28H;
call     LcdSendCommand

;Entry mode set, increment, no shift

movlw    06H;
call     LcdSendCommand

;Display ON, Curson ON, Blink OFF

movlw    0EH
call     LcdSendCommand

;Clear display

call     LcdClear

return

```

```

;*****
;
; Clear LCD
;*****

```

LcdClear

```

;Clear display

movlw    01H
call     LcdSendCommand

movlw    2                ;Wait 2 ms
call     msDelay

;DD RAM address set 1st digit

movlw    80H;
call     LcdSendCommand

return

```

```

;*****
;
; Locate cursor on LCD
; W = D7-D4 row, D3-D0 col
;*****

```

LcdLocate

```

movwf    tmpLcdRegister+0

movlw    80H
movwf    tmpLcdRegister+1

movf     tmpLcdRegister+0,W
andlw    0FH
iorwf    tmpLcdRegister+1,F

btfsc    tmpLcdRegister+0,4
bsf      tmpLcdRegister+1,6

movf     tmpLcdRegister+1,W
call     LcdSendCommand

return

```

```

;*****
;
; Send a data to LCD

```

```
;*****
;
```

LcdSendData

```
    bsf      PORTB, LCD_RS
    call     LcdSendByte
    return
```

```
;*****
;
; Send a command to LCD
;*****
```

LcdSendCommand

```
    bcf      PORTB, LCD_RS
    call     LcdSendByte
    return
```

```
;*****
;
; Send a byte to LCD by 4 bit data bus
;*****
```

LcdSendByte

```
    ;Save value to send

    movwf    tmpLcdRegister

    ;Send higher four bits

    bcf      PORTB, LCD_DB4
    bcf      PORTB, LCD_DB5
    bcf      PORTB, LCD_DB6
    bcf      PORTB, LCD_DB7

    btfsc    tmpLcdRegister, 4
    bsf      PORTB, LCD_DB4
    btfsc    tmpLcdRegister, 5
    bsf      PORTB, LCD_DB5
    btfsc    tmpLcdRegister, 6
    bsf      PORTB, LCD_DB6
    btfsc    tmpLcdRegister, 7
    bsf      PORTB, LCD_DB7

    bsf      PORTB, LCD_E      ;Enables LCD
    movlw    1                ;Wait 1ms
    call     msDelay
    bcf      PORTB, LCD_E      ;Disabled LCD
    movlw    1                ;Wait 1ms
    call     msDelay

    ;Send lower four bits

    bcf      PORTB, LCD_DB4
    bcf      PORTB, LCD_DB5
    bcf      PORTB, LCD_DB6
    bcf      PORTB, LCD_DB7

    btfsc    tmpLcdRegister, 0
    bsf      PORTB, LCD_DB4
    btfsc    tmpLcdRegister, 1
    bsf      PORTB, LCD_DB5
    btfsc    tmpLcdRegister, 2
    bsf      PORTB, LCD_DB6
    btfsc    tmpLcdRegister, 3
    bsf      PORTB, LCD_DB7

    bsf      PORTB, LCD_E      ;Enables LCD
    movlw    1                ;Wait 1ms
```

```

    call    msDelay
    bcf     PORTB, LCD_E      ;Disabled LCD
    movlw   1                 ;Wait 1ms
    call    msDelay

    return

```

```

;*****
;
; Send a character on RS232
; (9600 baud,8 data bit, 1 stop bit, No parity)
;
; Input: W = Character to send
;*****

```

TxChar

```

    movwf   ShiftReg

    movlw   8                 ;Data lenght
    movwf   BitCount

    bcf     PORTA, TX         ;Send start bit
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop

    DELAY   BIT_DELAY

    ;Tx loop

```

TxLoop

```

    btfss   ShiftReg, 0
    goto    TxLo

    nop
    bsf     PORTA, TX
    goto    cTx

```

TxLo

```

    bcf     PORTA, TX
    goto    cTx

```

cTx

```

    nop
    rrf     ShiftReg, F

    DELAY   BIT_DELAY

    decfsz  BitCount, F
    goto    TxLoop
    nop
    nop
    nop
    nop

    bsf     PORTA, TX          ;Stop bit
    DELAY   BIT_DELAY
    DELAY   2
    nop

    bsf     PORTA, TX
    DELAY   BIT_DELAY
    DELAY   2
    return

```

```

;*****
; Receive a character from RS232
; (9600 baud,8 data bit, 1 stop bit, No parity)
;
; Return code:
;
; ShiftReg: Received character
;*****

```

RxChar

```

    clrf    ShiftReg

    movlw   8                ;Data lenght
    movwf   BitCount

    DELAY   BIT_DELAY+BIT_DELAY/2    ;Wait 1.5 bit

    ;Loop di lettura dei bit dati

```

wDB

```

    btfss   PORTA,RX
    goto    RxBitL

```

RxBitH

```

    nop
    bsf     STATUS,C
    goto    RxShift

```

RxBitL

```

    bcf     STATUS,C
    goto    RxShift

```

RxShift

```

    nop
    rrf     ShiftReg,F

    DELAY   BIT_DELAY

    decfsz  BitCount,F
    goto    wDB
    return

    END

```

```

;*****
; Pic by example
; RS232IO.ASM
;
; (c) 2001, Sergio Tanzilli
; http://www.tanzilli.com
;*****

        PROCESSOR      16F84
        RADIX          DEC
        INCLUDE        "P16F84.INC"

;Suppress the following MPASM warning message [# 302]:
;"Register in operand not in bank 0.  Ensure that bank bits are correct"

        ERRORLEVEL     -302

;Flag configuration

        __CONFIG        3FF1H

;RS232 lines

TX      equ      0      ;Tx data
RX      equ      1      ;Rx data

;I/O lines on PORTB

LED1     equ      0
LED2     equ      1
LED3     equ      2
LED4     equ      3

SWITCH1  equ      4
SWITCH2  equ      5
SWITCH3  equ      6
SWITCH4  equ      7

;Command code from PC

LED1_ON  equ      00h
LED2_ON  equ      01h
LED3_ON  equ      02h
LED4_ON  equ      03h

LED1_OFF equ      10h
LED2_OFF equ      11h
LED3_OFF equ      12h
LED4_OFF equ      13h

GET_SWITCH equ      20h

;*****
; Clock frequency related constant (4 MHz)
;*****

BIT_DELAY equ      23      ;Bit delay a 9600 bps

;*****
; MACRO - Delay subroutine with watch dog timer clearing
;
; Macro parameters:
;
;   VALUE:      Delay obtained = ((VALUE-1)*4+5)*(1/(Fosc/4))
;
;*****

DELAY      MACRO      VALUE

```

```

LOCAL    REDO

movlw    VALUE
movwf    TmpRegister

REDO

    clrwdt                ;Clear watch dog timer

    decfsz TmpRegister,F
    goto  REDO

    ENDM

;*****
;
; FILE REGISTER
;*****

ORG      0CH

;Register used by msDelay subroutine and DELAY macro

msDelayCounter    res    2
TmpRegister       res    1

;Register used by RS232 subroutines

ShiftReg          res    1        ;Shift register
BitCount          res    1        ;Bit counter

DummyReg          res    1

;*****
;
; RESET VECTOR
;*****

ORG      00H

Start

    bsf      STATUS,RP0        ;Swap to register bank 1

    movlw    00011111B        ;Sets the whole PORTA as input
    movwf    TRISA

    movlw    11111111B        ;Sets the whole PORTB as input
    movwf    TRISB

    bcf      PORTA,TX          ;Sets TX line as output

    bcf      TRISB,LED1        ;Set output line on PORTB
    bcf      TRISB,LED2
    bcf      TRISB,LED3
    bcf      TRISB,LED4

    bcf      STATUS,RP0        ;Swap to register bank 0

    bcf      PORTB,LED1        ;Turn off each leds
    bcf      PORTB,LED2
    bcf      PORTB,LED3
    bcf      PORTB,LED4

MainLoop

    ;Wait until receives a start bit from RS232 line

    btfsc    PORTA,RX          ;Received a start bit ?
    goto    MainLoop          ;No, wait.

    call     RxChar            ;Yes, read the byte on receiving...

;*****
;
; Check for PC commands

```

```

; *****
;

```

```

; *****
;
; LED1_ON
; *****
;

```

Led10n

```

movlw    LED1_ON
xorwf    ShiftReg,W
btfss    STATUS,Z
goto     _Led10n

```

```

bsf      PORTB,LED1

```

```

goto     MainLoop

```

_Led10n

```

; *****
;
; LED2_ON
; *****
;

```

Led20n

```

movlw    LED2_ON
xorwf    ShiftReg,W
btfss    STATUS,Z
goto     _Led20n

```

```

bsf      PORTB,LED2

```

```

goto     MainLoop

```

_Led20n

```

; *****
;
; LED3_ON
; *****
;

```

Led30n

```

movlw    LED3_ON
xorwf    ShiftReg,W
btfss    STATUS,Z
goto     _Led30n

```

```

bsf      PORTB,LED3

```

```

goto     MainLoop

```

_Led30n

```

; *****
;
; LED4_ON
; *****
;

```

Led40n

```

movlw    LED4_ON
xorwf    ShiftReg,W
btfss    STATUS,Z
goto     _Led40n

```

```

bsf      PORTB,LED4

```

```

goto     MainLoop

```

_Led40n

```

; *****
;
; LED1_OFF
; *****
;

```

Led10ff

```

movlw    LED1_OFF
xorwf    ShiftReg,W
btfss    STATUS,Z
goto     _Led10ff

```

```
    bcf      PORTB, LED1
```

```
    goto     MainLoop
```

```
_Led10ff
```

```
    ;*****
    ;
    ; LED2_OFF
    ;*****
```

```
Led20ff
```

```
    movlw    LED2_OFF
    xorwf    ShiftReg,W
    btfss    STATUS,Z
    goto     _Led20ff
```

```
    bcf      PORTB, LED2
```

```
    goto     MainLoop
```

```
_Led20ff
```

```
    ;*****
    ;
    ; LED3_OFF
    ;*****
```

```
Led30ff
```

```
    movlw    LED3_OFF
    xorwf    ShiftReg,W
    btfss    STATUS,Z
    goto     _Led30ff
```

```
    bcf      PORTB, LED3
```

```
    goto     MainLoop
```

```
_Led30ff
```

```
    ;*****
    ;
    ; LED4_OFF
    ;*****
```

```
Led40ff
```

```
    movlw    LED4_OFF
    xorwf    ShiftReg,W
    btfss    STATUS,Z
    goto     _Led40ff
```

```
    bcf      PORTB, LED4
```

```
    goto     MainLoop
```

```
_Led40ff
```

```
    ;*****
    ;
    ; GET_SWITCH
    ;*****
```

```
GetSwitch
```

```
    movlw    GET_SWITCH
    xorwf    ShiftReg,W
    btfss    STATUS,Z
    goto     _GetSwitch
```

```
    swapf    PORTB,W           ;Read the switch state and send
    movwf    DummyReg         ;it to the PC
    comf     DummyReg,W
    andlw    0Fh
    call     TxChar
```

```
    goto     MainLoop
```

```
_GetSwitch
```

```
    goto     MainLoop
```

```

;*****
; Delay subroutine
;
; W = Requested delay time in ms (clock = 4MHz)
;*****

msDelay
    movwf    msDelayCounter+1
    clrf     msDelayCounter+0

    ; 1 ms (about) internal loop

msDelayLoop
    nop
    decfsz   msDelayCounter+0,F
    goto     msDelayLoop
    nop

    decfsz   msDelayCounter+1,F
    goto     msDelayLoop

    return

;*****
; Send a character on RS232
; (9600 baud,8 data bit, 1 stop bit, No parity)
;
; Input: W = Character to send
;*****

TxChar
    movwf    ShiftReg

    movlw    8                ;Data lenght
    movwf    BitCount

    bcf      PORTA,TX         ;Send start bit
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop

    DELAY    BIT_DELAY

    ;Tx loop

TxLoop
    btfss    ShiftReg,0
    goto     TxLo

    nop
    bsf      PORTA,TX
    goto     cTx

TxLo
    bcf      PORTA,TX
    goto     cTx

cTx
    nop
    rrf      ShiftReg,F

    DELAY    BIT_DELAY

    decfsz   BitCount,F
    goto     TxLoop
    nop

```

```

nop
nop
nop

bsf    PORTA, TX          ;Stop bit
DELAY  BIT_DELAY
DELAY  2
nop

bsf    PORTA, TX
DELAY  BIT_DELAY
DELAY  2
return

```

```

;*****
;
; Receive a character from RS232
; (9600 baud,8 data bit, 1 stop bit, No parity)
;
; Return code:
;
;   ShiftReg:  Received character
;*****

```

RxChar

```

    clrf    ShiftReg

    movlw   8          ;Data lenght
    movwf   BitCount

    DELAY   BIT_DELAY+BIT_DELAY/2    ;Wait 1.5 bit

    ;Loop di lettura dei bit dati

```

wDB

```

    btfss   PORTA, RX
    goto    RxBitL

```

RxBitH

```

    nop
    bsf     STATUS, C
    goto    RxShift

```

RxBitL

```

    bcf     STATUS, C
    goto    RxShift

```

RxShift

```

    nop
    rrf     ShiftReg, F

    DELAY   BIT_DELAY

    decfsz  BitCount, F
    goto    wDB
    return

END

```



Descrizione dei pin del PIC16F84

1**RA2**

E' una linea di I/O programmabile in ingresso o in uscita dall'utente.

Corrisponde al **BIT 2** della **PORTA A**.

2**RA3**

E' una linea di I/O programmabile in ingresso o in uscita dall'utente.

Corrisponde al **BIT 3** della **PORTA A**.

3**RA4 / RTCC**

E' un pin multifunzione che può essere programmato come normale linea di I/O oppure come linea di clock in ingresso verso il contatore RTCC.

Se programmato come linea di I/O corrisponde al **BIT 4** della **PORTA A** diversamente dalle altre linee di I/O, quando questa linea funziona come uscita, lavora a collettore aperto.

4**MCLR / VPP**

In condizioni di normale funzionamento svolge le funzioni di **Master CLeaR** ovvero di [Reset](#) ed è attivo a livello 0. Può essere collegato ad un circuito esterno di reset o più semplicemente collegato fisso al positivo.

Quando il PIC viene posto in "Program Mode" viene utilizzato come ingresso per la tensione di programmazione Vpp.

5**VSS**

E' il pin a cui va connesso il **negativo** della tensione di alimentazione.

6**RB0**

E' una linea di I/O programmabile in ingresso o in uscita dall'utente.

Corrisponde al **BIT 0** della **PORTA B** e può essere programmata per generare interrupt.

7**RB1**

E' una linea di I/O programmabile in ingresso o in uscita dall'utente.

Corrisponde al **BIT 1** della **PORTA B**.

8**RB2**

E' una linea di I/O programmabile in ingresso o in uscita dall'utente.

Corrisponde al **BIT 2** della **PORTA B**.

9	RB3 E' una linea di I/O programmabile in ingresso o in uscita dall'utente. Corrisponde al BIT 3 della PORTA B .
10	RB4 E' una linea di I/O programmabile in ingresso o in uscita dall'utente. Corrisponde al BIT 4 della PORTA B .
11	RB5 E' una linea di I/O programmabile in ingresso o in uscita dall'utente. Corrisponde al BIT 5 della PORTA B .
12	RB6 E' una linea di I/O programmabile in ingresso o in uscita dall'utente. Corrisponde al BIT 6 della PORTA B .
13	RB7 E' una linea di I/O programmabile in ingresso o in uscita dall'utente. Corrisponde al BIT 7 della PORTA B .
14	VDD E' il terminale positivo di alimentazione del PIC. In tutte e tre le versioni disponibili del PIC16F84 (commercial, industrial e automotive) la tensione può assumere un valore che va da un minimo di 2.0 volt ad un massimo di 6.0 volt .
15	OSC2 / CLKOUT E' un pin di connessione nel caso venga utilizzato un quarzo per generale il clock. E' l'uscita del clock nel caso venga applicato un oscillatore RC o un oscillatore esterno.
16	OSC1 / CLKIN E' un pin di connessione nel caso venga utilizzato un quarzo o un circuito RC per generare il clock. E' l'ingresso del clock nel caso venga utilizzato un oscillatore esterno.
17	RA0 E' una linea di I/O programmabile in ingresso o in uscita dall'utente. Corrisponde al BIT 0 della PORTA A .

18

RA1

E' una linea di I/O programmabile in ingresso o in uscita dall'utente.

Corrisponde al **BIT 1** della **PORTA A**.



Set di istruzioni del PICmicro 16F84

Sintassi	Descrizione Microchip	Operazione equivalente
ADDLW k	Add Literal and W	$W = W + k$
ADDWF f,d	Add W and f	$d = W + f$ (dove d può essere W o f)
ANDLW k	AND Literal with W	$W = W \text{ AND } k$
ANDWF f,d	AND W with f	$d = W \text{ AND } f$ (dove d può essere W o f)
BCF f,b	Bit Clear f	$f(b) = 0$
BSF f,b	Bit Set f	$f(b) = 1$
BTFSC f,b	Bit Test f, Skip if Clear	$f(b) = 0$? Sì, salta una istruzione
BTFSS f,b	Bit Test f, skip if Set	$f(b) = 1$? Sì, salta una istruzione
CALL k	Subroutine Call	Chiama la subroutine all'indirizzo k
CLRF f	Clear f	$f = 0$
CLRW	Clear W Register	$W = 0$
CLRWDI	Clear Watchdog Timer	Watchdog timer = 0
COMF f,d	Complement f	$d = \text{not } f$ (dove d può essere W o f)
DECF f,d	Decrement f	$d = f - 1$ (dove d può essere W o f)
DECFSZ f,d	Decrement f, Skip if 0	$d = f - 1$ (dove d può essere W o f) se $d = 0$ salta
GOTO k	Go to address	Salta all'indirizzo k
INCF f,d	Increment f	$d = f + 1$ (dove d può essere W o f)
INCFSZ f,d	Increment f, Skip if 0	$d = f + 1$ (dove d può essere W o f) se $d = 0$ salta
IORLW k	Inclusive OR Literal with W	$W = W \text{ OR } k$
IORWF f,d	Inclusive OR W with f	$d = f \text{ OR } W$ (dove d può essere W o f)
MOVLW k	Move literal to W	$W = k$
MOVF f,d	Move f	$d = f$ (dove d può essere W o f)
MOVWF f	Move W to f	$f = W$
NOP	No Operation	Nessuna operazione
OPTION	Load Option Register	$\text{OPTION} = W$
RETFIE	Return from Interrupt	Ritorna da un interrupt handler



RETLW k	Return Literal to W	Ritorna da una subroutine con $W = k$
RETURN	Return from Subroutine	Ritorna da una subroutine
RLF f,d	Rotale Left f through Carry	$d = f \ll 1$ (dove d può essere W o f)
RRF f,d	Rotale Right f through Carry	$d = f \gg 1$ (dove d può essere W o f)
SLEEP	Go into Standby Mode	Mette in standby il PIC
SUBLW k	Subtract W from Literal	$W = k - W$
SUBWF f,d	Subtract W from f	$d = f - W$ (dove d può essere W o f)
SWAPF f	Swap f	f = Swap dei bit 0123 con 4567 di f
TRIS f	Load TRIS Register	TRIS di f = W
XORLW k	Exclusive OR Literal with W	$W = W \text{ XOR } k$
XORWF f,d	Exclusive OR W with f	$d = f \text{ XOR } W$ (dove d può essere W o f)

ADDLW**ADD Literal and W****Somma la costante k a W****Sintassi:**

```
addlw    k
```

Operazione equivalente:
$$W = W + k$$
Descrizione:

Somma la costante **k** al valore memorizzato nell'accumulatore **W** e mette il risultato nell'accumulatore.

Esempio:

```
        org      00H

start:  movlw    10
        addlw    12
        . . .
```

Dopo aver eseguito questo programma l'accumulatore W varrà 22.

Note:

Questa istruzione influenza i bit **Z**, **DC** e **C** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.
- **DC** vale 1 se il risultato dell'operazione è un numero superiore a 15.
- **C** vale 1 se il risultato è positivo ovvero se il bit 7 del registro contenente il risultato vale 0 e 1 se il risultato è negativo ovvero se il bit 7 del registro contenente il risultato vale 1.

ADDWF**ADD W and F**

Somma il valore contenuto in W con il valore contenuto nel registro F

Sintassi:

addwf **f, d**

Operazione equivalente:

$d = W + f$ (dove d può essere W o f)

Descrizione:

Questa istruzione somma il valore contenuto nell'accumulatore **W** con il valore contenuto nel registro indirizzato dal parametro **f**. Il parametro **d** è un flag che indica su quale registro deve essere memorizzato il risultato.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Esempio:

Vediamo un esempio di somma tra due registri:

```

add1    equ    0CH
add2    equ    0DH

           org    00H

           movlw   10           ;Primo addendo = 10
           movwf   add1

           movlw   15           ;Secondo addendo = 15
           movwf   add2

           movf    add1, W      ;W = add1
           addwf   add2, W      ;W = W + add2
  
```

Note:

Questa istruzione influenza i bit **Z**, **DC** e **C** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.
- **DC** vale 1 se il risultato dell'operazione è un numero superiore a 15.
- **C** vale 1 se il risultato è positivo ovvero se il bit 7 del registro contenente il risultato vale 0 e 1 se il risultato è negativo ovvero se il bit 7 del registro contenente il risultato vale 1.

ANDLW**AND Literal with W****Effettua l'AND tra W ed una costante k****Sintassi:**

```
andlw    k
```

Operazione equivalente:
$$W = W \text{ AND } k$$
Descrizione:

Effettua l'AND tra il valore contenuto nell'accumulatore **W** ed il valore costante **k**. Il risultato viene memorizzato nell'accumulatore.

Esempio:

```
                org      00H

start
    movlw       10101010B
    andlw       11110000B
    . . .
```

Dopo aver eseguito questo programma l'accumulatore W varrà 10100000B.

Note:

Questa istruzione influenza il bit **Z** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.

ANDWF**AND W with F****Effettua l'AND logico tra il valore contenuto in W ed il valore contenuto nel registro F****Sintassi:****andwf** **f, d****Operazione equivalente:** $d = W \text{ AND } f$ (dove d può essere W o f)**Descrizione:**

Questa istruzione effettua l'AND logico tra il valore contenuto nell'accumulatore **W** ed il valore contenuto nel registro indirizzato dal parametro **f**. Il parametro **d** è un flag che indica su quale registro deve essere memorizzato il risultato.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Esempio:

Spesso l'AND logico viene utilizzato per mascherare il valore di alcuni bit all'interno di un registro. Se ad esempio volessimo estrarre dal numero binario 01010101B i quattro bit meno significativi al fine di ottenere il seguente valore 00000101B, basterà preparare una maschera del tipo 00001111B e farne l'AND con il nostro valore di partenza, vediamo come:.

```

movlw    01010101B    ;Memorizza nel registro                ; all'indirizzo
movwf    0CH           ;0CH il valore iniziale da mascherare

movlw    00001111B    ;Prepara la maschera di bit
andwf    0CH,W         ;Effettua l'AND e memorizza il
                       ;risultato nell'accumulatore W

```

Il risultato in W sarà 00000101B come richiesto.

```
W = 00001111 AND
f = 01010101 =
-----
W = 0000010101
```

La **ANDWF** influenza il bit **Z** del registro **STATUS** che varrà 1 se il risultato dell'operazione è 0.

BCF**Bit Clear F****Azzera un bit nel registro F****Sintassi:**

```
bcf    f, b
```

Operazione equivalente:
$$f(b) = 0$$
Descrizione:

Questa istruzione azzera il **bit b** del registro all'indirizzo **f**.

Esempio:

```
parm1    equ    0CH
          org    00H
          movlw    11111111B    ;Valore iniziale
          movwf    parm1
          bcf      parm1, 0      ;D0=0
```

Al termine del programma il registro parm1 varrà **11111110B**.

Note:

Questa istruzione non influenza alcun bit di stato

BSF**Bit Set F****Mette a uno un bit nel registro F****Sintassi:**

```
bsf    f, b
```

Operazione equivalente:
$$f(b) = 1$$
Descrizione:

Questa istruzione mette a uno il **bit b** del registro all'indirizzo **f**.

Esempio:

```
parm1  equ    0CH
        org    00H
        movlw  00000000B  ;Valore iniziale
        movwf  parm1
        bsf    parm1, 0    ;D0=1
```

Al termine del programma il registro parm1 varrà **00000001B**.

Note:

Questa istruzione non influenza alcun bit di stato

BTFSK**Bit Test F, Skip if Clear****Salta l'istruzione successiva se un bit nel registro F vale 0****Sintassi:****btfsc** **f, b****Operazione equivalente:**

f(b) = 0 ? Sì, salta una istruzione

Descrizione:

Testa il bit b contenuto nel registro all'indirizzo f e salta l'istruzione successiva se questo vale 0.

Esempio:

```

parm1    equ        0CH

          org        00H

          movlw    11111110B    ;Valore iniziale
          movwf    parm1

loop
          btfsc    parm1, 0        ;D0 = 0 ? Sì, esce
          goto     loop            ;No, esegue il loop

```

Questa programma esegue un loop infinito lo stesso programma non esegue il loop se sostituiamo l'istruzione:

```

movlw    11111110B    ;Valore iniziale

```

con l'istruzione:

```

movlw    11111111B    ;Valore iniziale

```

Note:

Questa istruzione non influenza alcun bit di stato

BTFSS**Bit Test F, Skip if Set****Salta l'istruzione successiva se un bit nel registro F vale 1****Sintassi:****btfss** **f, b****Operazione equivalente:**

f(b) = 1 ? Sì, salta una istruzione

Descrizione:

Testa il bit b contenuto nel registro all'indirizzo f e salta l'istruzione successiva se questo vale 1.

Esempio:

```

parm1    equ        0CH

          org        00H

          movlw    11111111B    ;Valore iniziale
          movwf    parm1

loop
          btfss    parm1, 0        ;D0 = 1 ? Sì, esce
          goto     loop            ;No, esegue il loop

```

Questo programma esegue un loop infinito lo stesso programma non esegue il loop se sostituiamo l'istruzione:

```

movlw    11111111B    ;Valore iniziale

```

con l'istruzione:

```

movlw    11111110B    ;Valore iniziale

```

Note:

Questa istruzione non influenza alcun bit di stato

CALL**Subroutine CALL****Chiamata a subroutine****Sintassi:**

```
call    k
```

Descrizione:

Richiama in esecuzione una [subroutine](#) memorizzata all'indirizzo **k**. Il parametro k può essere specificato utilizzando direttamente il valore numerico dell'indirizzo oppure la relativa label.

Esempio:

```
org      00H

call     ledOn
...

;Subroutine di accensione di un led

ledOn
    bsf    PORTB, LED1
    return
```

Quando la CPU del PIC incontra una istruzione CALL, memorizza nello STACK il valore del registro PC + 1 in modo da poter riprendere l'esecuzione dall'istruzione successiva alla CALL, quindi scrive nel PC l'indirizzo della subroutine saltando all'esecuzione di quest'ultima.

Il valore originale del PC viene ripristinato all'uscita della subroutine con l'esecuzione dell'istruzione di ritorno [RETURN](#) o [RETLW](#).

Nel PIC16F84 sono disponibili 8 livelli di stack, per cui il numero massimo di CALL rientranti, ovvero di istruzioni CALL all'interno di subroutine che a loro volta contengono altre CALL, è limitato ad 8 livelli.

Note:

Questa istruzione non influenza nessun bit di stato.

CLRF**CLear F** register**Azzerare il registro F****Sintassi:****clrf** **f****Operazione equivalente:****f = 0****Descrizione:**

Questa istruzione azzerare il valore contenuto nel registro indirizzato dal parametro **f**.

Esempio:

Ipotizziamo di voler azzerare il registro **TMR0** il cui indirizzo è 01H esadecimale, l'istruzione da eseguire sarà:

clrf **01H**

Oppure, se si include all'inizio del nostro source il file [P16F84.INC](#), potremo utilizzare il nome simbolico del registro **TMR0**.

clrf **TMR0**

Dopo l'esecuzione di questa istruzione il bit **Z** del registro **STATUS** viene messo a **1**.

CLRW**CLear W register****Azzera il registro W****Sintassi:****clrw****Operazione equivalente:** $W = 0$ **Descrizione:**

Azzera il valore contenuto nel registro **W**.

Note:

Dopo l'esecuzione di questa istruzione il bit **Z** del registro **STATUS** viene messo a **1**.

CLRWDT**CLear WatchDog Timer****Reset del timer del watchdog****Sintassi:****clrwdt****Descrizione:**

Questa istruzione deve essere utilizzata quando programmiamo il PIC con l'opzione Watchdog abilitata (fusibile WDTE). In questa modalità il PIC abilita un timer che, una volta trascorso un determinato tempo, effettua il reset del PIC. Per evitare il reset il nostro programma dovrà eseguire ciclicamente l'istruzione CLRWDT per azzerare il timer prima di detto tempo. Se non azzeriamo il timer in tempo, la circuiteria di watchdog (dall'inglese cane da guardia) interpreterà questo come un blocco del programma in esecuzione ed effettuerà il reset per sbloccarlo.

Esempio:

```
                org      00H

loop           clrwdt
               goto    loop
```

Note:

Questa istruzione non influenza nessun bit di stato.

COMF**COMplement F****Effettua il complemento del registro F****Sintassi:****comf** **f, d****Operazione equivalente:**

d = NOT f (dove d può essere W o f)

Descrizione:

Questa istruzione effettua il complemento del valore contenuto nel registro indirizzato dal parametro **f**. Il parametro **d** determina la destinazione del valore ottenuto.

Per **d = W** il valore viene memorizzato nel **registro W**

Per **d = F** il valore viene lasciato nel **registro f**.

Esempio:

```

parm1    equ      0CH

          org      00H

          movlw    01010101B
          movwf    parm1

          comf     parm1, F
          ...

```

Al termine dell'esecuzione del programma il valore del registro **parm1** sarà **10101010B**.

Note:

Questa istruzione influenza il bit **Z** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.

DECF**DECrement F register****Azzera il contenuto del registro F****Sintassi:****decf** **f, d****Operazione equivalente:** $d = f - 1$ (dove d può essere W o f)**Descrizione:**

Questa istruzione decrementa il contenuto del registro indirizzato dal parametro **f**. Il parametro **d** è un flag che indica su quale registro deve essere memorizzato il risultato.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Esempio:

Con il seguente programma scriviamo il valore 23H nel registro all'indirizzo 0CH e quindi lo decrementiamo di uno. Al termine dell'esecuzione il registro alla locazione 0CH conterrà il valore 22H.

```

movlw    23H    ;Scrivo in W il valore 23H
movwf    0CH    ;Copia nel registro 0CH il valore di W
decf     0CH,F  ;Decremento il valore contenuto nel
                ;registro 0CH

```

Esempio:

Questa istruzione influenza il bit **Z** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.

DECFSZ**DECrement F, Skip if Zero**

Decrementa il valore del registro f e salta l'istruzione successiva se il risultato vale zero

Sintassi:

```
decfsz  f, b
```

Operazione equivalente:

$d = f - 1$ (dove d può essere W o f) se $d = 0$ salta

Descrizione:

Decrementa il valore del registro all'indirizzo **f** e se il risultato vale zero salta l'istruzione successiva. Il risultato del decremento può essere memorizzato nello stesso registro **f** oppure nell'accumulatore **W** in base al valore del flag **d**.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Esempio:

```
counter equ    0CH
org          00H
movlw       10          ;counter = 10
movwf      counter
loop
  decfsz    counter, F  ;counter = counter -1
                        ;counter = 0 ? Si esce
  goto     loop         ;No, continua
```

Questa programma esegue per 10 volte l'istruzione decfsz finchè esce per counter = 0.

Note:

Questa istruzione non influenza alcun bit di stato.

GOTO**GO TO** address**Vai in esecuzione all'indirizzo k****Sintassi:**

```
goto    k
```

Descrizione:

Determina un salto del programma in esecuzione all'indirizzo k. Il parametro k può essere specificato utilizzando direttamente il valore numerico dell'indirizzo oppure la relativa label.

Esempio:

```
        org    00H  
  
loop  
        goto   loop
```

Questo programma esegue un ciclo (loop) infinito.

Note:

Questa istruzione non influenza nessun bit di stato.

INCF**INC**rement **F****Incrementa il valore del registro all'indirizzo F****Sintassi:****incf** **f, d****Operazione equivalente:** $d = f + 1$ (dove d può essere W o f)**Descrizione:**

Incrementa il contenuto del registro all'indirizzo **f** e memorizza il risultato nello stesso registro o nell'accumulatore **W** in base al valore del flag **d**:

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nello stesso **registro F**

Note:

Questa istruzione influenza il bit **Z** del registro **STATUS**..

- **Z** vale 1 se il risultato dell'operazione vale 0.

INCFsz**INCrement F, Skip if Zero**

Incrementa il valore del registro **f** e salta l'istruzione successiva se il risultato vale zero

Sintassi:

```
incfsz  f, b
```

Operazione equivalente:

$d = f + 1$ (dove d può essere W o f) se $d = 0$ salta

Descrizione:

Incrementa il valore del registro all'indirizzo **f** e se il risultato vale zero salta l'istruzione successiva. Il risultato dell'incremento può essere memorizzato nello stesso registro **f** oppure nell'accumulatore **W** in base al valore del flag **d**.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Esempio:

```
counter equ    0CH

org           00H

movlw        250           ;counter = 250
movwf        counter

loop
    incfsz    counter, F    ;counter = counter + 1
                        ;counter = 0 ? Si esce
    goto      loop         ;No, continua
```

Questa programma esegue per $256 - 10 = 6$ volte l'istruzione `incfsz` finchè esce per `counter = 0`. Essendo `counter` un registro a 8 bit quando viene incrementato dal valore 255 assume nuovamente valore 0 da qui la formula $256 - 10 = 6$.

Note:

Questa istruzione non influenza alcun bit di stato.

IORLW**Inclusive OR Literal with W****Effettua l'OR inclusivo tra W ed una costante k****Sintassi:**

```
iorlw    k
```

Operazione equivalente:
$$W = W \text{ OR } k$$
Descrizione:

Effettua l'OR inclusivo tra il valore contenuto nell'accumulatore **W** ed il valore costante **k**.

Esempio:

```
        org        00H

start:  movlw       00001111B
        iorlw       11110000B
        . . .
```

Dopo aver eseguito questo programma l'accumulatore W varrà **11111111B**.

Note:

Questa istruzione influenza il bit **Z** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.

IORWF**Inclusive OR W with F**

Effettua l'OR inclusivo tra il valore contenuto in W ed il valore contenuto nel registro F

Sintassi:

```
iorwf    f, d
```

Operazione equivalente:

$d = f \text{ OR } W$ (dove d può essere W o f)

Descrizione:

Questa istruzione effettua l'OR inclusivo tra il valore contenuto nell'accumulatore **W** ed il valore contenuto nel registro indirizzato dal parametro **f**. Il parametro **d** determina dove viene memorizzato il risultato dell'operazione:

Per **d = W** il risultato viene memorizzato nell'accumulatore **W**.

Per **d = F** il risultato viene memorizzato nel registro **f**.

Esempio:

```
parm1    equ    0CH

          org    00H

          movlw   00001111B
          movwf   parm1

          movlw   11111111B
          iorwf   parm1, F
```

Al termine dell'esecuzione il valore del registro parm1 sarà **11111111B**.

Note:

Questa istruzione influenza il bit **Z** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.

MOVLW**MOV**e Literal to **W****Assegna a W un valore costante****Sintassi:**

```
movlw    k
```

Operazione equivalente:
$$W = k$$
Descrizione:

Assegna all'accumulatore **W** il valore costante **k**.

Esempio:

```
                org    00H  
  
start  
    movlw    20  
    . . .
```

Dopo aver eseguito questo programma l'accumulatore W varrà 20.

Note:

Questa istruzione non influenza nessun bit di stato.

MOVF**MOVe F****Muove il contenuto del registro F****Sintassi:****movf** **f, d****Operazione equivalente:**

d = f (dove d può essere W o f)

Descrizione:

Questa istruzione copia il contenuto del registro indirizzato dal parametro **f** o nell'accumulatore **W** o nello stesso **registro F**. Il parametro **d** determina la destinazione.

Per **d = W** il valore viene memorizzato nel **registro W**

Per **d = F** il valore viene lasciato nel **registro f**. In questo caso l'utilità dell'istruzione sta nel fatto che viene alterato il bit Z del flag STATUS in base al valore contenuto nel **registro f**.

Esempio:

L'esempio seguente copia il valore contenuto nel registro all'indirizzo 0CH nell'accumulatore W:

movf **0CH, W**

MOVWF**MOVe W to F****Muove il contenuto del registro W nel registro F****Sintassi:****movwf** **f****Operazione equivalente:****f = W****Descrizione:**

Questa istruzione copia il contenuto del **registro W** nel registro indirizzato dal parametro **f**.

Esempio:

Ipotizziamo di voler scrivere il valore **10H** ([esadecimale](#)) nel registro **TMR0**. Le istruzioni da eseguire sono le seguenti.

```
movlw    10H        ;Scrivo nel registro W il valore 10H
movwf    01H        ;e lo memorizzo nel registro TMR0
```

Per i registri utilizzati dal PIC per funzioni specifiche, solitamente non viene inserito direttamente l'indirizzo ma il relativo nome simbolico definito nel file [P16F84.INC](#). Il codice diventa quindi il seguente:

```
movlw    10H        ;Scrivo nel registro W il valore 10H
movwf    TMR0       ;e lo memorizzo nel registro TMR0
```

Note:

L'esecuzione della **MOVWF** non influenza nessun bit di stato.

NOP**No OPeration****Nessuna operazione****Sintassi:****nop****Descrizione:**

Questa istruzione non esegue nessuna operazione ma è utile per inserire ritardi pari ad un ciclo macchina .

Esempio:

Utilizzando un quarzo da **4MHz** potremo ottenere un ritardo pari ad **1µs** per ogni istruzione **NOP** inserita nel nostro source..

nop ;Esegue un ritardo pari ad 1µs**Note:**

La **NOP** non influenza nessun bit di stato.

OPTIONload **OPTION** register**Assegna il valore in W al registro OPTION****Sintassi:****option****Operazione equivalente:**

OPTION = W

Descrizione:

Questa istruzione memorizza nel registro speciale OPTION il valore contenuto nell'accumulatore W.

Esempio:

```

org      00H

start
    movlw 01000100B
    option
    . . .

```

Note:

Questa istruzione esiste per mantenere la compatibilità con i PIC prodotti finora, la Microchip ne sconsiglia l'uso. In alternativa è consigliabile usare le seguenti istruzioni.

```

org      00H

start

    bsf    STATUS,RP0 ;Attiva il banco registri 1

    movlw  01000100B
    movwf  OPTION_REG
    . . .

```

In pratica si consiglia di scrivere direttamente nel registro OPTION presente nel banco 1 dei registri del PIC utilizzando la MOVWF anziché l'istruzione OPTION che in futuro potrebbe non essere più implementata.

Questa istruzione non influenza nessun bit di stato.

RETFIE**RET From Interrupt****Ritorna da una subroutine****Sintassi:**

```
retfie
```

Descrizione:

Questa istruzione deve essere inserita al termine di ogni subroutine di gestione degli interrupt per ridare il controllo al programma principale.

Esempio:

```
                org      00H
loop            goto     loop    ;Loop infinito

                org      04H    ;Interrupt vector

intHandler
                retfi        ;Ritorna dall'interrupt
```

In questo source il programma principale esegue un loop infinito. Se abilitiamo uno degli interrupt del 16F84 non appena esso si verificherà il controllo verrà dato automaticamente al programma allocato dall'indirizzo 04H (nell'esempio intHandler), l'istruzione RETFI determinerà quindi il ritorno al loop principale.

Note:

Questa istruzione non influenza alcun bit di stato.

RETLW**RETurn Literal to W****Ritorna da una subroutine con una costante in W****Sintassi:**

```
retlw    k
```

Descrizione:

Questa istruzione ritorna il controllo da una subroutine al programma principale. A differenza dell'istruzione [RETURN](#) essa consente di passare, tramite l'accumulatore **W**, il valore costante **k** al programma principale.

Esempio:

```
rtc      equ      0CH

          org      00H
          call     mySub1
          movwf    rtc

          . . .

mySub1:
          nop
          retlw    10
```

Una volta eseguito questo programma memorizza nel registro rtc il valore 10 passato dalla subroutine mySub1.

Note:

Questa istruzione non influenza alcun bit di stato

Vedi anche l'istruzione [RETURN](#).

RETURN**RETURN** from subroutine

Ritorna da una subroutine

Sintassi:**return****Descrizione:**

Questa istruzione deve essere inserita al termine di ogni subroutine per riprendere l'esecuzione del programma principale.

Esempio:

```

                org      00H
                call     mySub1
                ....
mySub1:
                nop
                return

```

Note:

Nel PIC16F84 possono essere annidate fino ad 8 chiamate a subroutine del tipo:

```

                org      00H
                call     mySub1
                ....
mySub1:
                call     mySub2
                return
mySub2:
                call     mySub3
                return
mySub3:
                return

```

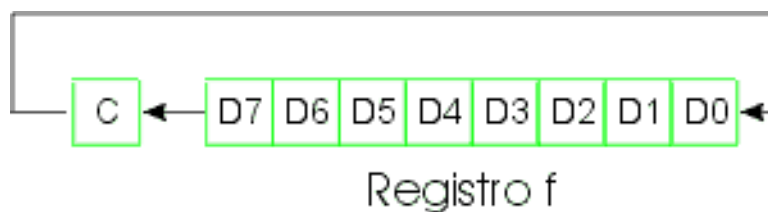
Note:

Questa istruzione non influenza alcun bit di stato.

Vedi anche l'istruzione [RETLW](#).

RRF**Rotate Left F through carry****Ruota a sinistra il contenuto del registro f passando per il Carry****Sintassi:****rlf** **f, b****Operazione equivalente:** $d = f \ll 1$ (dove d può essere W o f)**Descrizione:**

Ruota i bit contenuti nel registro all'indirizzo **f** verso sinistra (ovvero dai bit meno significativi verso quelli più significativi) passando per il bit **CARRY** del registro **STATUS** come illustrato in figura:



Il contenuto del bit CARRY del registro status viene spostato nel bit D0 mentre il valore in uscita dal bit D7 viene spostato nel CARRY.

Il valore del parametro **d** determina la destinazione del risultato ottenuto al termine della rotazione:

Per **d = W** il risultato viene memorizzato nel **registro W** lasciando il registro **f** invariato.

Per **d = F** il risultato viene memorizzato nello stesso **registro f**

Esempio:

```

parm1    equ      0CH

           org      00H

clrf     C, STATUS    ;Azzera il CARRY

movlw    01010101B    ;Valore iniziale
movwf    parm1

rlf      parm1, F

```

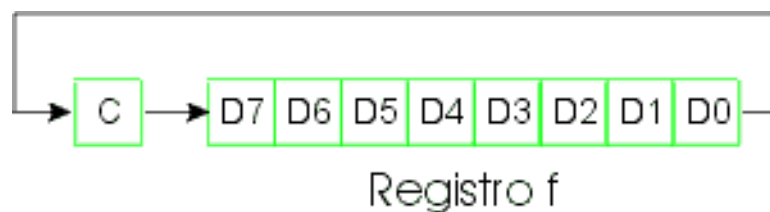
Al termine del programma il registro **parm1** varrà **10101010B** mentre il CARRY varrà 0.

Note:

Questa istruzione non influenza nessun altro bit di stato oltre al CARRY.

RRF**Rotate Right F through carry****Ruota a destra il contenuto del registro f passando per il Carry****Sintassi:****rrf** **f, b****Operazione equivalente:** $d = f \gg 1$ (dove d può essere W o f)**Descrizione:**

Ruota i bit contenuti nel registro all'indirizzo **f** verso destra (ovvero dai bit più significativi verso quelli meno significativi) passando per il bit **CARRY** del registro **STATUS** come illustrato in figura:



Il contenuto del bit CARRY del registro status viene spostato nel bit D7 mentre il valore in uscita dal bit D0 viene spostato nel CARRY.

Il valore del parametro **d** determina la destinazione del risultato ottenuto al termine della rotazione:

Per **d = W** il risultato viene memorizzato nel **registro W** lasciando il registro f invariato.

Per **d = F** il risultato viene memorizzato nello stesso **registro f**

Esempio:

```

parm1    equ      0CH

           org      00H

clrf     C, STATUS    ;Azzera il CARRY

movlw    01010101B    ;Valore iniziale
movwf    parm1

rrf      parm1, F

```

Al termine del programma il registro **parm1** varrà **00101010B** mentre il CARRY varrà 1.

Note:

Questa istruzione non influenza nessun altro bit di stato oltre al CARRY.

SLEEP**go into standby mode****Mette il PIC in standby****Sintassi:****sleep****Descrizione:**

Questa istruzione blocca l'esecuzione del programma in corso e mette il PIC in stato di standby (sleep dall'inglese to sleep, dormire).

Esempio:

```
org      00H  
  
start  
    sleep
```

Note:

Questa istruzione non influenza nessun bit di stato.

SUBLW**SUB**tract **W** from **L**iteralSottrae a **k** il valore in **W****Sintassi:**

```
sublw    k
```

Operazione equivalente:

$$W = k - W$$

Descrizione:

Sottra alla costante **k** il valore memorizzato nell'accumulatore **W**.

Esempio:

```
                org      00H  
  
start  
    movlw      10  
    sublw      12  
    . . .
```

Dopo aver eseguito questo programma l'accumulatore W varrà 2.

Note:

Questa istruzione influenza i bit **Z**, **DC** e **C** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.
- **DC** vale 1 se il risultato dell'operazione è un numero superiore a 15.
- **C** vale 1 se il risultato è positivo ovvero se il bit 7 del registro contenente il risultato vale 0 e 1 se il risultato è negativo ovvero se il bit 7 del registro contenente il risultato vale 1.

SUBWF**SUB**stract **W** from **F**

Sottrae il valore contenuto in **W** dal valore contenuto nel registro **F**

Sintassi:

```
subwf    f, d
```

Operazione equivalente:

$d = f - W$ (dove d può essere W o f)

Descrizione:

Questa istruzione sottrae il valore contenuto nel **registro W** dal valore contenuto nel registro indirizzato dal parametro **f**. Il parametro **d** è un flag che indica su quale registro deve essere memorizzato il risultato.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Esempio:

Analizziamo un esempio estratto dal datasheet della Microchip:

Se inseriamo l'istruzione:

```
subwf    REG1, F
```

Dove **reg1** è l'indirizzo di un qualsiasi registro specificato con la direttiva:

```
REG1     RES      1
```

Per valori iniziali di $REG1=3$ e $W=2$, dopo l'esecuzione avremo $REG1=1$ e $C=1$ in quanto il risultato è positivo.

Per valori iniziali di $REG1=2$ e $W=2$ dopo l'esecuzione avremo $REG1=0$ e $C=1$ perché il risultato è sempre positivo.

Per valori iniziali di $REG1=1$ e $W=2$, avremo $REG1=FFH$ ovvero -1 quindi $C=0$ perché il risultato è negativo.

Note:

Questa istruzione influenza i bit **Z**, **DC** e **C** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.
- **C** vale 1 se il risultato è positivo ovvero se il bit 7 del registro contenente il risultato vale 0 e 1

se il risultato è negativo ovvero se il bit 7 del registro contenente il risultato vale 1.

SWAPF**SWAP F**

Scambia il valore dei quattro bit più significativi del registro all'indirizzo F con i quattro bit meno significativi

Sintassi:

swap **f, d**

Operazione equivalente:

f = Swap dei bit 0123 con 4567 di f

Descrizione:

Scambia il valore dei quattro bit più significativi (D7-D4) contenuti nel registro all'indirizzo f con i quattro bit meno significativi (D3-D0) dello stesso. Il risultato viene memorizzato nell'accumulatore **W** o nello stesso registro **f** in base al valore di **d**:

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nello stesso **registro F**

Note:

Questa istruzione non influenza alcun bit di stato

TRISload **TRIS** register

Assegna il valore in W al registro TRIS

Sintassi:`tris f`**Operazione equivalente:**

TRIS di f = W

Descrizione:

Questa istruzione memorizza in uno dei registri speciale TRIS il valore contenuto nell'accumulatore W. I registri TRIS determinano il funzionamento in ingresso e uscita delle linea di I/O del PIC. Esiste un registro TRIS per ogni porta di I/O denominato TRISA, TRISB, ecc.

Esempio:

```

org      00H

start

movlw    11111111B
tris     PORTA
...

```

Note:

Questa istruzione esiste per mantenere la compatibilità con i PIC prodotti finora, la Microchip ne sconsiglia l'uso. In alternativa è consigliabile usare le seguenti istruzioni.

```

org      00H

start

bsf      STATUS,RP0      ;Attiva il banco registri 1

movlw    11111111B
movwf    TRISA
...

```

In pratica si consiglia di scrivere direttamente nei registri TRIS presenti nel banco 1 dei registri del PIC utilizzando la MOVWF anziché l'istruzione TRIS che in futuro potrebbe non essere più implementata.

Note:

Questa istruzione non influenza nessun bit di stato.

XORLW**Exclusive OR Literal with W****Effettua l'OR esclusivo tra W ed una costante k****Sintassi:**

```
xorlw    k
```

Operazione equivalente:
$$W = W \text{ XOR } k$$
Descrizione:

Effettua l'OR esclusivo tra il valore contenuto nell'accumulatore **W** ed il valore costante **k**.

Esempio:

```
                org      00H

start
    movlw      00000000B
    xorlw      11110000B
    . . .
```

Dopo aver eseguito questo programma l'accumulatore W varrà 11110000B.

Note:

Questa istruzione influenza il bit **Z** del registro **STATUS**.

- **Z** vale 1 se il risultato dell'operazione vale 0.

XORWF**eXclusive OR W with F**

Effettua l'OR esclusivo tra il valore contenuto in W ed il valore contenuto nel registro F

Sintassi:

```
xorwf    f, d
```

Operazione equivalente:

$d = f \text{ XOR } W$ (dove d può essere W o f)

Descrizione:

Questa istruzione effettua l'OR esclusivo (XOR) tra il valore contenuto nell'accumulatore **W** ed il valore contenuto nel registro indirizzato dal parametro **f**. Il parametro **d** è un flag che indica su quale registro deve essere memorizzato il risultato.

Per **d = W** il risultato viene memorizzato nel **registro W**

Per **d = F** il risultato viene memorizzato nel **registro f**

Questa istruzione influenza i bit **Z** del registro **STATUS** che varrà 1 se il risultato dell'operazione è 0.

Esempio:

Ipotizziamo di dover effettuare lo XOR tra il registro W ed il registro **REG1** da noi definito all'indirizzo **0CH** con la direttiva:

```
REG1    EQU    0CH
```

possiamo utilizzare l'istruzione **IORWF** in due forme a seconda di dove vogliamo mettere il risultato, ovvero:

```
xorwf    COUNTER, F    ; COUNTER = COUNTER XOR W
```

oppure:

```
xorwf    COUNTER, W    ; W = COUNTER XOR W
```

Note:

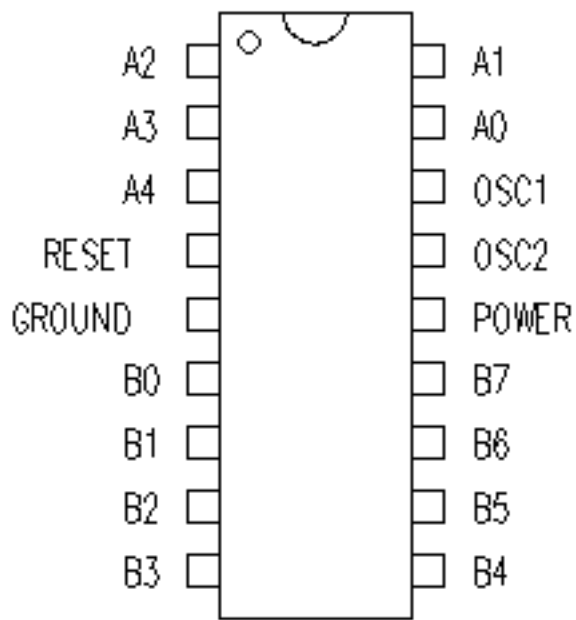
L'OR esclusivo (XOR) è un'operazione tra due bit in cui il bit risultante vale 0 se i due bit sono uguali.

Spesso lo XOR viene utilizzato nell'assembler del PIC per effettuare la comparazione tra due valori in mancanza di un'istruzione specifica.

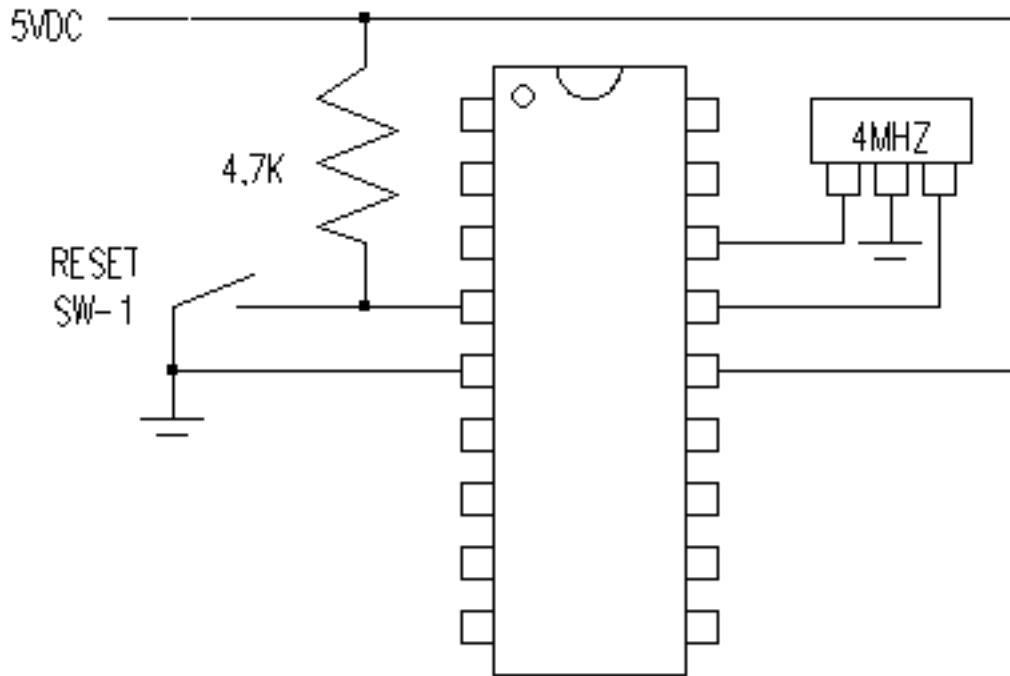
Vediamo come:

ipotizziamo di avere un valore nel registro REG1 e di voler verificare se è uguale a 57H. Le istruzioni da eseguire sono le seguenti:

<u>movlw</u>	57H	;W = Valore da comparare = 57H ;Risultato. W = 57H
xorwf	REG1,W	;W = W XOR REG1 Effettua lo XOR con ;il valore in REG1
<u>btfss</u>	STATUS,Z	;Salta l'istruzione seguente se il ;risultato dello XOR vale 0, ovvero ;se il valore di REG1 e' pari a 57H
<u>goto</u>	diverso	;Salta se diverso da 57H
<u>goto</u>	uguale	;Salta se uguale da 57H



PIC.16F84-4



PIC.16F84-4

