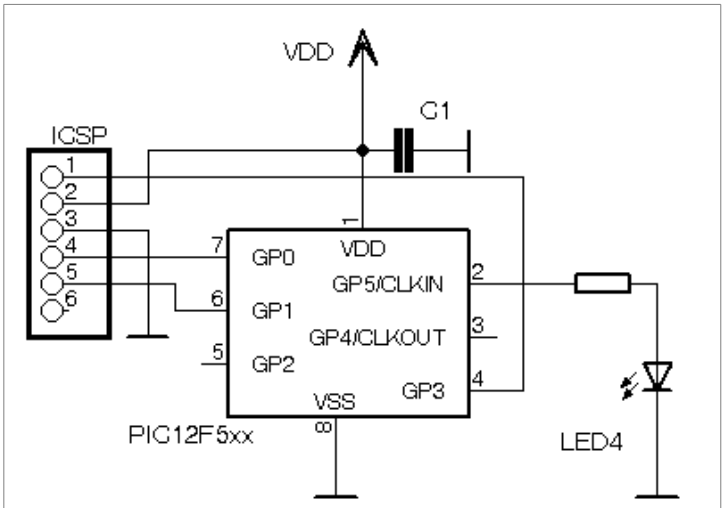
Esercitazioni PIC Baseline

2A - Assembly - LED lampeggiante : subroutine.

Cosa vogliamo ottenere

Lo scopo dell' esercitazione è quello di fare lampeggiare un LED attraverso il microcontroller.

Utilizziamo sempre un **12F519** (ma, come in precedenza, va bene qualsiasi PIC, con gli opportuni adattamenti al sorgente - qui saranno proposte le variazioni per 12F508/509 , 10F200/202 e 16F505/526 .



Lo schema elettrico è lo stesso dell'esercitazione precedente.

Il LED è collegato tra il pin **GP5** del microcontroller e la massa (Vss): configurando il pin come uscita digitale e mandandolo a livello 1 (che è pari circa alla tensione positiva di alimentazione, Vdd), il LED si accenderà.

Portando il livello a 0, il LED sarà spento.

I pin dei PIC possono erogare 25mA; quindi, limitando la corrente nel LED con una resistenza in serie, non occorre alcun altro oggetto esterno.

Questi collegamenti sono già presenti sulle demo board, mentre se usiamo una breadboard vanno eseguiti una volta per tutte, dato che questo schema di base sarà utilizzato in gran parte delle altre esercitazioni.
Per la LPCminiBoard i collegamenti sono quelli visti nell' esercitazione precedente.

Anche questa esercitazione è "densa" , contenendo vari elementi e concetti importanti:

- **uso della memoria RAM**
- **creazione e uso di subroutine**

Sono state introdotte alcune pagine di chiarimento relative alle subroutines nei Baseline e alle pagine della memoria, argomenti che vengono qui introdotti e che saranno sviluppati progressivamente.

Il sorgente - Temporizzazioni

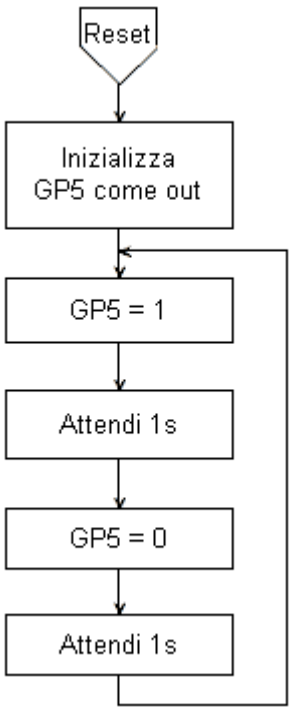
Come sempre, il primo passo nella realizzazione di un qualsiasi programma è quello avere ben chiaro cosa si vuole ottenere :

- **accendere il LED collegato al `GP5` non appena viene data tensione e farlo lampeggiare alla frequenza di 1 Hz circa fino a che la tensione è presente**

e come ottenerlo:

- **programmando il pin `GP5` come uscita digitale e portandolo a livello alto per 1 secondo (LED acceso), poi spegnendolo (livello basso) per un altro secondo e così via (loop indefinito).**

Ora sarà possibile tracciare un diagramma di flusso (*flow chart*) che mostra in forma grafica quanto dovrà essere eseguito dal programma. Si potrebbe pensare che esso non sia importante, ma, se ora è cosa da poco, può diventare la chiave che permette al programmatore di risolvere situazioni altrimenti troppo complesse per una azione di analisi solamente mentale.



Il diagramma riporta la sequenza delle operazioni da eseguire.

Un errore comune ai principianti è quello di non valutare correttamente i tempi di esecuzione delle istruzioni del processore.

Con il clock a 4MHz, le istruzioni si susseguono al ritmo di una ogni microsecondo (milionesimo di secondo), per cui, nel lampeggio del LED, occorre implementare una certa attesa tra le commutazioni on/off per permettere all' occhio di percepire la variazione di stato.

Al reset, `GP5` viene programmato come uscita digitale e portato a livello alto (1) e il LED si accende.

Il programma effettua un ritardo di 1 secondo, quindi spegne il LED portando l' uscita digitale a 0. Ora un' altra attesa di 1 secondo, quindi il programma effettua un loop ripetendo l' accensione del LED e le fasi successive indefinitamente, fino a che il circuito è alimentato.

Con l'immagine data dal diagramma a blocchi diventa molto più semplice scrivere le istruzioni necessarie; altra facilitazione nella stesura del sorgente è l' utilizzo del **template** già visto**.**
Anche in questo esempio il sorgente è reso disponibile già completamente scritto, nel file ***2A_519.asm*** (**2A_5089.asm** per i PIC12F508/509).
Nonostante questo, è necessario che non lo accettiamo così come sta, ma ne facciamo una analisi dettagliata, in modo da vere ben chiare tutte le sue componenti ed il perchè delle varie scelte. Quindi possiamo stamparlo e seguire la spiegazione dei vari passi.
Come al solito, stampiamo il sorgente e vediamone i dettagli.

Inizio e definizioni di base sono le stesse dell' esercizio precedente, con le sole modifiche del caso e non sono necessari altri commenti.

Dobbiamo aggiungere una struttura di istruzioni (algoritmo) per fare "attendere" il processore un certo tempo

prima di passare all' operazione successiva; questo può essere semplicemente ottenuto con un algoritmo "**waste time**".
Il metodo è quello di sfruttare il tempo impiegato dal ciclo di una istruzione che non abbia alcuna attività al di fuori del processore, ad esempio un **nop**.

NOP

Questa istruzione non svolge alcun lavoro (**nop** = ***no operation***), ma richiede un ciclo di clock per la sua esecuzione; quindi, essenzialmente, "consuma" tempo senza fare altro.
Il suo scopo è quello di creare un "tempo vuoto da azioni", che spesso si rende necessario per temporizzate correttamente le operazioni di I/O.

La sua sintassi è:

[label]	sp	nop	sp	[;commento]

La struttura della riga è quella generale delle righe di istruzione.

- l' opcode non può iniziare in prima colonna
- la riga può iniziare con una label opzionale
- la riga può terminare con un commento opzionale

A differenza di altri opcodes, **nop** non ha un oggetto.

Loop di tempo

Con un clock 4MHz la frequenza per il ciclo istruzione è:

Fosc/4 = 4MHz / 4 = 1MHz

e il periodo relativo è **1 us**. Quindi, per ottenere 1 secondo occorreranno 1000000 di questi cicli.
Certamente scrivere:

```
; attesa 1000000 di cicli
  nop
  nop
  nop
 .....
```

ripetendo l' istruzione per 1 milione di volte è semplice, ma non è pensabile, anche solo perchè non ci sono disponibili 1 milione di locazioni di memoria programma (ogni istruzione occupa una locazione...).
Un sistema più evoluto è quello di utilizzare un contatore pre caricato con il valore voluto e decrementarlo fino al suo azzeramento, facendo così passare il numero voluto di microsecondi:

```
; attesa 1000000 di cicli
       movlw   .333333      ; inizializza counter
       movwf   contatore
loop   decfsz  contatore,f  ; counter = 0 ?
        goto   loop         ; no - altro loop
fineconteggio ...           ; si - fine attesa
```

Il funzionamento è il seguente:

- `movlw .333333` carica il registro **W** con il numero (decimale) indicato
- `movwf contatore` copia il contenuto di **W** nel registro `contatore`
- `loop decfsz contatore,f` il contenuto di `contatore` viene decrementato di una unità; se il risultato è diverso da zero viene eseguita la riga seguente `goto loop` che fa eseguire un nuovo decremento
- se il risultato del decremento è zero, la riga `goto` viene saltata e il programma prosegue con il successivo `fineconteggio ...`

Ne risulta che il programma viene arrestato ad eseguire il loop 333333 + 1 volte. Vediamo il perché.

Dato che la riga `decfsz contatore,f` impiega 1 ciclo (1us @ 4MHz) per essere eseguita e la riga `goto loop` impiega 2 cicli (2us @ 4MHz), il loop impiega 333333 x (2+1) = 999999 us.
Quando il contatore è azzerato, la `decfsz` richiede 1 ciclo in più per il salto della riga successiva, per cui il tempo totale sarà 999999 + 1 = 1000000us, ovvero il secondo voluto.

Questo sistema è uno dei più semplici algoritmi "**waste time**", tecnica impiegata comunemente per ottenere un ritardo durante l' esecuzione del programma.

Se può sembrare strana la necessità di avere microcontroller che lavorano a 4, 8, 20, 40, 60 e più MHz per poi far loro "perdere tempo", dobbiamo convincerci che, invece, si tratta di una necessità primaria in quanto i processi che il chip deve controllare sono assai spesso legati a elementi meccanici o elettrici o, come in questo caso, necessità di comunicare all'operatore una segnalazione visiva, che hanno tempi di risposta molto lenti, se comparati alla velocità di esecuzione delle istruzioni. Ne risulta la necessità di avere nello stesso tempo un processore veloce per poter seguire con sicurezza tutte le operazioni legate al controllo, come ad esempio calcoli matematici, e, nello stesso tempo, la necessità di adeguare le azioni di controllo alle caratteristiche dei componenti che vengono controllati.

In questo senso, il "waste time" (tempo buttato, letteralmente) non ha alcuna valenza morale, ma è un elemento essenziale del funzionamento del programma. In relazione alla frequenza di clock, è da notare che il principiante accetta acriticamente quanto viene proposto durante la didattica, in quanto marginale. Ma al momento delle progettazione di un circuito, è elemento importante e che determina la possibilità o meno di eseguire determinate operazioni. Oltre al fatto che il consumo energetico del processore è proporzionale alla frequenza del clock.

DECFSZ

L' istruzione `decfsz` (*decrement file and skip next instruction if zero*) svolge le seguenti azioni:

- diminuisce di 1 il registro `contatore`
- e salta l' istruzione successiva se il risultato è 0

Con l'opzione `f` si specifica che il risultato del decremento va conservato nel file stesso.

 In molte istruzioni è possibile specificare se il risultato, ad esempio di un decremento o di un incremento, oppure di una operazione logica, debba essere conservato nel registro (**,f**) oppure passato direttamente a W (**,w**). Queste opzioni rendono estremamente funzionale l' impiego di queste istruzioni.

L' istruzione fa sì che, quando il contenuto del contatore è a zero**, il *Program Counter* salti l' opcode**

seguente: non viene eseguito quello immediatamente successivo, ma quello seguente.

Nella sequenza:

```
loop    decfsz  contatore,f
         goto   loop
        goto    endloop
```

fino a che contatore è > 0, la sequenza è:

```
loop    decfsz  contatore,f
         goto   loop
```

arrivato a zero il contatore, la riga `goto loop` **viene saltata** e si passa alla successiva `goto endloop`, ponendo fine al loop.

 Attenzione**:** l'istruzione, quando il contatore è azzerato, non fa saltare il programma ad una certa destinazione, come farebbe una istruzione di jump condizionato di cui dispongono altri processori, **ma fa saltare al *Program Counter* solo l' opcode successivo**.
Questo deve essere ben chiaro quando si usa questa istruzione.

E' evidente la funzione complessiva della sequenza, che permette di contare un certo numero di passi. Il contatore è pre caricato con un determinato valore, in questo caso 333333, in quanto il loop ripetuto è composto da `decfsz` e dal `goto` che impiegano rispettivamente 1 e 2 cicli per essere eseguite (333333 x 3us = 1s). L' istruzione è estremamente comoda all'interno di loop in cui occorre diminuire il valore di un contatore e verificare nel contempo quando questo si è azzerato. La sintassi dell' istruzione è la solita degli opcodes di Microchip:

[label]	sp	decfsz	, f / w	sp	oggetto	sp	[;commento]

La label iniziale può essere omessa, se non serve; l' opcode, come solito, non può comunque iniziare in prima colonna.
Osserviamo, però, che l' opcode **deve** avere una "coda" che definisce la destinazione del file che viene mosso:

- `decfsz,w` indica al compilatore che il risultato del decremento andrà copiato in W
- `decfsz,f` indica al compilatore che il risultato del decremento andrà copiato sul file stesso

Questa è una caratteristica delle istruzioni di Microchip e consente una notevole flessibilità e potenza.

Nel primo caso, il file in oggetto viene decrementato di una unità e il risultato viene posto nel registro W, senza modificare il file stesso.
Ovvero, se il file contiene 12h , al termine della istruzione si avrà:

- **file = 12h**
- **W = 11h**

Nel secondo caso, il file in oggetto viene decrementato di una unità e il risultato viene posto nel file stesso, mentre il registro W non è modificato
Ovvero, se il file contiene **12h** , al termine della istruzione si avrà:

- **file = 11h**

- **W non modificato**

Questo consente di elaborare implementazioni di vari algoritmi con il minimo impiego di opcodes (e quindi di spazio in memoria programma e di velocità di esecuzione).

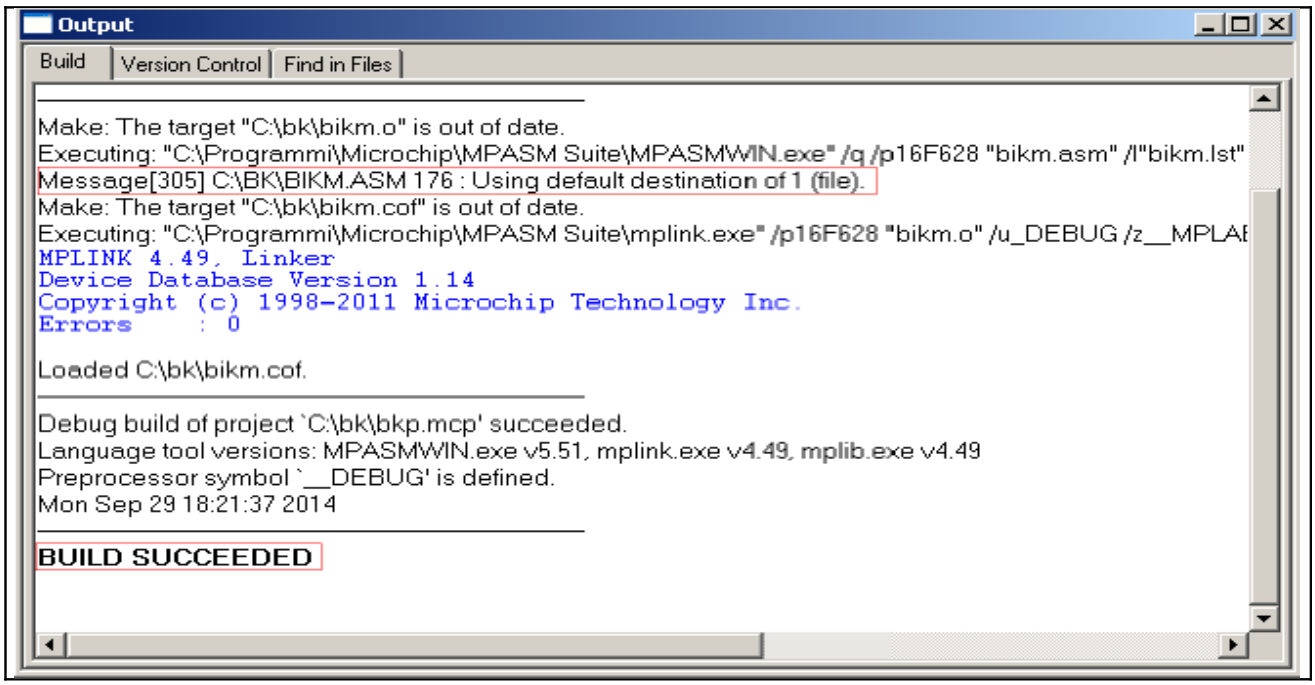
`decfsz` non è la sola istruzione che richiede una "coda"; ne vedremo altre nel corso delle varie esercitazioni. Per tutte valgono le regole generali indicate qui.

ATTENZIONE : è arrivato un Message…

Se trascuriamo di scrivere la "coda" all'istruzione, l' **Assembler MPASM genera un Message**, ovvero un avviso al programmatore.
L' avviso indica che la compilazione viene portata a termine ugualmente, **ma che è stata usata la "coda" di default, ovvero `,f` .**

In effetti, sfugge spesso di completare gli opcodes che richiedono la "coda" ed un richiamo da parte del compilatore è necessario.
L'azione di correzione è pure sensata; se così non fosse, il compilatore, mancando un elemento dell' opcode, dovrebbe abortire l' intera compilazione. In pratica, dato che, nella maggioranza dei casi, il risultato dell' operazione è destinato al file stesso, i progettisti hanno deciso di introdurre questa correzione automatica, **avvisando comunque il programmatore in modo che possa intervenire nel caso in cui la "coda" doveva essere diversa**.
Message è la terza categoria di segnalazioni, assieme a **Error** e **Warning** , che l' Assembler MPASM invia all' utente al termine della compilazione.

Un Message è una informazione su qualcosa che non pregiudica la compilazione stessa, ma di cui il programmatore deve essere messo al corrente. Troviamo **Error**, **Warning** e **Message** elencati nella finestra ***Output***, nella cartella ***Build***., al termine della compilazione. Ecco un esempio:



Osserviamo che è presente la riga **`Message [305]`** che ci indica la correzione effettuata.
La correzione non blocca la compilazione e viene generato il file ***.hex***: lo vediamo dal messaggio **`BUILD SUCCEEDED`**.

Questo, però, avviene anche se la logica impostata nelle istruzioni avrebbe richiesto di terminare l' opcode con un `,w` invece di un `,f` !!!

Quindi, nonostante questo automatismo, **è vivamente sconsigliabile scrivere l'opcode incompleto**, a scanso di errori, difficili poi da diagnosticare, e che porterebbero al non funzionamento del programma. E soprattutto:

Non trascuriamo di leggere il contenuto della finestra Output al termine della compilazione .
Non è sufficiente ricevere il rassicurante messaggio **`BUILD SUCCEEDED`** per considerare riuscita la compilazione, ma occorre anche verificare che non siano presenti **Message** o **Warning** di cui si debba tener conto.

Leggere i messaggi forniti dal compilatore non è una inutile opzione, ma una necessità.

,F ,W

Si può ancora notare che, dal punto di vista dei valori assoluti, esiste l' equivalenza :

`,f`	`,1`
`,w`	`,0`

ovvero la scrittura :

```
decfsz,w
decfsz,f
```

è identica a :

```
decfsz,0
decfsz,1
```

Anche se si vede spesso usare quest'ultima forma, essa **è sconsigliabile**.
MPASM riconosce la "coda" simbolica della prima notazione, quindi non è necessario ricorrere a numeri.

Inoltre, come evidente, la prima forma è auto esplicativa, mentre la seconda non lo è e può creare facilmente confusione ed errori.

333333 ?

Tutto questo sarebbe semplice se non ci fosse un limite insuperabile: l' ampiezza massima di un dato, in questi microcontroller, è **8 bit**, per cui il numero massimo caricabile è **255** (**FFh**) e la linea **`movlw .333333`** è impossibile !

Se scrivessimo questa riga, il compilatore segnalerebbe una anomalia:

```
Warning[202] C:\2A_519.ASM 68 : Argument out of range. Least significant bits used.
```

Ovvero, il compilatore ha trovato un numero troppo grande per essere contenuto nel registro W e per default ne utilizza solamente i bit più bassi.

Dato che si tratta di un Warning, la compilazione viene portata a termine e appare la scritta: **`BUILD SUCCEEDED.`** Questo è abbastanza grave in quanto il programma non funzionerà come ci si aspetta., ovvero, ancora una volta:

Non trascuriamo di leggere il contenuto della finestra Output al termine della compilazione .
Non è sufficiente ricevere il rassicurante messaggio **`BUILD SUCCEEDED`** per considerare riuscita la compilazione, ma occorre anche verificare che non siano presenti **Message** o **Warning** di cui si debba tener conto.

Esiste ovviamente una soluzione, che è la seguente:

- dato che abbiamo a che fare con un numero che non può essere contenuto in un solo registro a 8 bit, occorre utilizzare più di un contatore, definendo diverse locazioni di memoria RAM
- queste locazioni saranno pre caricate con opportuni valori
- e vengono decrementate eseguendo loop di istruzioni concatenate.

Quando il contenuto di tutti i contatori è esaurito, sono stati consumati i cicli voluti

Il calcolo di un simile algoritmo non è complicato, ma richiede calcoli lunghi e noiosi, basati sulla durata delle istruzioni, con la possibilità di sbagliare. Nella pratica non è necessario impegnare tempo per questo compito, dato che la rete offre un certo numero di "calculators" ad hoc, tra i quali il più serio è quello al sito di **Nikolai Golovchenko**. Ecco come è risolta l' attesa di 1 secondo con un clock di 4MHz:

```
; Delay = 1 secondo
; Clock frequency = 4 MHz
; Actual delay = 1 secondo = 1000000 cicli
Delay1s:                  ; 999997 cicli
  movlw   0x08            ; inizializza counters
  movwf   d1
  movlw   0x2F
  movwf   d2
  movlw   0x03
  movwf   d3
Delay1s_0:                ; loop di conteggio -<-|
  decfsz  d1, f           ;       d1=d1-1        |
   goto   $+2             ;->--|                 |
  decfsz  d2, f           ;    |  d2=d2-1        |
   goto   $+2             ;-<>-|                 |
  decfsz  d3, f           ;    |  d3=d3-1        |
   goto   Delay1s_0       ;-<>-|----->>----->>---|

; fine conteggio 999997 cicli - ora somma i 3 mancanti
  goto    $+1             ;->--|  3 cycles
  nop                     ;-<--|
```

La procedura è intricata, ma facilmente comprensibile con un po' di attenzione:

- i contatori **`d1,d2,d3`** sono caricati usando **`movlw`** (*move literal to W*) con valori fissi pre calcolati (**`movlw`** copia nel registro W il valore numerico –literal- indicato). I valori indicati fanno si che la

sequenza di decrementi impegni 999997 cicli.
La coppia di istruzioni:

```
movlw valore
movwf destinazione
```

costituisce la sequenza tipica per caricare in un determinato registro RAM (file) un valore numerico., dove l' istruzione **movwf** (*move W to file*) copia il contenuto di W nel file (locazione di RAM) indicato.

- il loop di conteggio decrementa il primo contatore **d1**, con l'opzione**,f** che specifica come il risultato del decremento va conservato nel file stesso. In altre parole si effettua l' operazione **d1 = d1 - 1**.
  Quando il contenuto è a zero, il Program Counter salta l' opcode seguente e quindi non viene eseguito questo codice, ma quello successivo.

- il Program Counter passa alla istruzione seguente fino a che il contenuto del file **d1** è maggiore di 0. La linea **goto $+2** avanza il program counter di due locazioni e lo porta sul successivo **goto $+2** che a sua volta lo porta sul **goto Delay1s_0** , che chiude il loop. Ogni istruzione **goto** impiega 2 cicli e ogni istruzione **decfsz** impiega 1 ciclo se consiste nel solo decremento (e 2 cicli se si effettua il salto).
  Ricordiamo che le istruzioni dei Baseline occupano una sola locazione di memoria programma ciascuna e quindi il calcolo con il simbolo **$** non riserva sorprese.

- quando **d1** è azzerato, l' istruzione fa saltare la linea **goto $+2** e passa alla **decfsz d2, f** , decremento di **d2** analogo al precedente. Si determina una altro loop che si esaurisce all' azzeramento del contatore **d2**.

- Inizia quindi il decremento di **d3**, al cui esaurimento istruzione **decfsz d3,f** fa saltare la linea di chiusura del loop **goto Delay1s_0** e gli pone fine. E' stato consumato un tempo pari a 999997 cicli.

- ora vengono aggiunti i tre cicli mancanti, realizzati con un **goto $+1** (che impegna 2 cicli) e un **nop** (1 ciclo). 1000000 cicli da 1us ciascuno sono esauriti ed è trascorso 1 secondo.

Da osservare che la procedura ha una label **Delay1s** come ingresso, che si dichiara "da se" essendo posta in prima colonna: il compilatore la verifica e se non ci sono problemi (duplicazione, errore di sintassi, uso di label riservata) la inserisce nella lista delle label dichiarate.

 Troviamo l'elenco delle label considerate dal compilatore in fondo al file di lista ***2A_519.lst*** generato dalla compilazione. Editando questo file possiamo renderci conto della complessità del lavoro svolto dall' Assembler**.**

All' interno della procedura, un' altra label, **Delay1s_0** , viene dichiarata per fare da "aggancio" al loop principale ed è richiamata dal **goto Delay1s_0** .

Per evitare di dichiarare altre label per i piccoli loop intermedi, si utilizza la scrittura **$** che indica il valore del program counter al momento attuale. Dato che è abbastanza facile incontrare esempi di programmazione che fanno uso di questo simbolo, è opportuno spendere due parole per descriverne la funzione e, sopratutto, i limiti.

Il simbolo $

Il segno **$** deriva dalle prime versioni di Assembler utilizzate per la programmazione dei computer e indica il valore corrente del ***Program Counter***.

Lo scopo originale per cui è nato questo segno è quello di ridurre il numero delle label dichiarate, calcolando manualmente di quante posizioni è lontana la destinazione, ad esempio per un salto.
Quindi, invece di scrivere:

```
        goto    next
next    nop
```

che richiede la dichiarazione della label , scrivo:

```
        goto    $+1
        nop
```

Questo perchè in passato la potenza di calcolo e la disponibilità di memoria erano basse e un elevato numero di label avrebbe rallentato o reso impossibile la compilazione.
Da tempo non esistono più questi problemi con i personal computer e quindi l'uso del simbolo è obsoleto, anche se tuttora possibile.
Il problema essenziale del suo uso è la difficoltà di calcolare il valore della lunghezza del salto, in quanto non tutti gli opcode possono avere la stessa lunghezza: se nei Baseline la lunghezza tipica è 1 byte, in altre famiglie la situazione è diversa, oltre al fatto che, in presenza di macro, il calcolo diventa problematico.

Qui usiamo **$** per la semplicità del tratto di programma in cui è inserito e che non riserva sorprese

Può essere di interesse consultare le pagine relative all' **uso del segno $**.

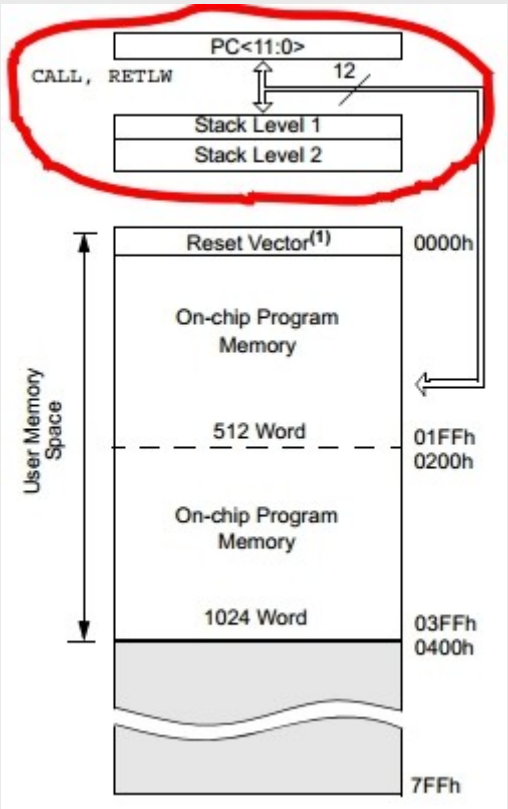
E un cenno allo stack

Il termine **stack** si indica un'area di memoria che viene usata dal microcontroller per immagazzinare gli indirizzi di rientro da subroutine, indirizzi che sono stati inseriti automaticamente dall' istruzione **`call`** e che, altrettanto automaticamente, sono automaticamente recuperati dall' istruzione **`return`** o **`retlw`**.

Lo stack opera come una memoria **LIFO** (***L**ast **I**n **F**irst **O**ut* – l' ultimo elemento aggiunto è il primo ad uscire), ovvero: i dati vengono estratti in ordine inverso rispetto a quello in cui sono stati inseriti.

Il nome, infatti, in inglese, indica una pila di oggetti, ad esempio piatti: è evidente che quando poniamo un piatto nella pila, lo si metta in cima (*top*) e che quando si preleva un piatto, lo si fa analogamente, partendo dal top.

Il primo elemento inserito nella pila è quello che si trova più in basso (*bottom*) e sarà anche l' ultimo ad essere estratto.



Un elemento significativo è dato dal fatto che, se in altre famiglie lo stack assume una certa ampiezza, anche dinamica, nei Baseline di Microchip è minimale, composto solamente da **due** "locazioni", dette livelli, e quindi in grado di contenere solamente due indirizzi.

Per i Baseline si tratta di due elementi nell' area della memoria con 12 bit di ampiezza ciascuno, ovvero in grado di contenere un indirizzo completo. Essi, pur essendo RAM, non sono accessibili in alcun modo all'utente

La limitazione a due soli livelli ha come conseguenza che non è possibile effettuare più di due chiamate a subroutine "annidate".

Questo è dovuto al fatto che lo stack non può contenere più degli elementi per cui è previsto e, se, riempita la pila, cerchiamo di aggiungere un' altro elemento, si determina un fatto molto grave: il primo indirizzo inserito, quello alla base o bottom, viene "cacciato fuori" dalla pila e va perduto.

In altre parole, per uno stack con solo due elementi (livelli), se eseguiamo una **`call`**, l' indirizzo di ritorno va nel primo livello dello stack (bottom). Al **`retlw`**, esso sarà estratto ed passato al Program Counter.
Se, prima del **`retlw`** effettuiamo una ulteriore **`call`**, il relativo indirizzo di ritorno si "appoggerà" sul primo, in cima alla pila (secondo livello).

L' azione della logica interna che, non appena l' ALU incontra un opcode **`call`** carica l' indirizzo di ritorno nello stack e a seguito di un **`retlw`**, scarica il top dello stack al *Program Counter*, è un processo del tutto automatico, che, nei Baseline, non è minimamente alterabile dall' utente.

Anche se in altri processori è possibile inserire o prelevare o manipolare elementi all' interno della pila con particolari istruzioni (*push, pop*), queste funzioni non sono presenti nei piccoli PIC, in cui lo stack è gestito esclusivamente dagli automatismi dell'unità centrale.

Nel caso in cui, prima di scaricare lo stack già riempito con i due indirizzi possibili, effettuiamo ancora una **`call`**, il suo indirizzo di ritorno viene messo al top e questo sovrascrive un indirizzo precedentemente immagazzinato. Ne risulta che una delle istruzioni di rientro da subroutine rimane priva del relativo indirizzo di rientro, dato che troverà lo stack vuoto, trasferendo al ***Program Counter*** un valore casuale. Questo non fa rientrare la subroutine al punto voluto, ma, nel caso migliore, al vettore di reset, se non ad un indirizzo casuale, creando seri problemi al funzionamento del programma.

Questa situazione è denominata ***stack owerflow*** (traboccamento dello stack). Dato che i Baseline non dispongono di meccanismi di protezione da questa situazione, sta al programmatore curare con attenzione di evitare di superare le due chiamate contemporaneamente attive e di inserirne ulteriori solamente dopo il completamento delle precedenti.

Come nota aggiuntiva, possiamo dire che i Midrange e i PIC a 8 bit superiori ai Baseline, hanno uno stack a 8 livelli, il che consente una elasticità di uso sufficiente per gran parte delle applicazioni; già il vetusto 16F84A (Midrange) aveva questa caratteristica ed il programmatore necessitava di minor attenzione a questo argomento, cosa che, con i Baseline, deve essere, invece, tenuta ben presente.

Di più, nei PIC18F sono implementati elementi per maneggiare lo stack, oltre a sistemi di protezione dall'overflow. In genere, i limiti dello stack e della sua gestione si riflettono sull' uso di linguaggi ad alto livello, dove esso viene impiegato intensivamente.

Un'ultima nota riguarda l' uso della parola stack per indicare in senso generale aree di memoria organizzate logicamente in modo analogo a quanto abbiamo visto ora. Ne deriva che si parla di ***stack Ethernet*** o ***stack USB*** per indicare quei buffer, ricavati dalla memoria RAM, attraverso complessi algoritmi, e necessari al funzionamento di questi protocolli di comunicazione.

Quanto è preciso il ritardo?

Da notare che l'algoritmo di tempo usato è stato studiato per generare una attesa molto precisa, cosa utile in altre applicazioni, ma che qui non è indispensabile, in quanto una cadenza di 1 secondo o 1 secondo e un decimo non farebbe gran differenza nel nostro caso.

Altri algoritmi del genere waste time sono realizzabili in numerosissime maniere, possibili anche utilizzando i timer integrati nel microcontroller. Qui abbiamo utilizzato quelle proposte dal sito indicato, anche se non strettamente necessarie per la loro precisione; questo perchè è possibile trovarsi in casi in cui le temporizzazioni sono impiegate comunicazioni seriali o RTC o comunque situazioni in cui va ricercata una elevata precisione nella generazione del tempo. Gli algoritmi indicati consentono questo, oltre al fatto del comodo applet in rete per il calcolo automatico dei parametri.

 Ovviamente **i tempi ottenuti da una struttura del genere sono precisi quanto lo è l' oscillatore che determina il clock delle istruzioni.**
Precisioni elevate si potranno avere solamente utilizzando clock altrettanto precisi.

Dato che la durata del ciclo è strettamente dipendente dalla frequenza del clock, se la frequenza si dimezza, il tempo si raddoppia. Se la frequenza raddoppia, ad esempio 8 MHz invece di 4, il ritardo sarà dimezzato.

Con 4MHz di clock il ciclo è Fosc/4= 1 MHz, ovvero 1 us di periodo. Con 8 MHz di clock, il ciclo è Fosc/4= 2 MHz, con periodo 500us.
Di conseguenza, per avere lo stesso ritardo con clock diversi, o per avere valori di ritardo differenti, occorrerà caricare i contatori con valori iniziali differenti, in modo da "consumare" il numero di cicli voluti.
Attualmente stiamo operando con il clock interno, che è un RC a 4MHz, ma di buona precisione (1%) e, come abbiamo visto, calibrabile. Se si desiderasse una precisione maggiore, ma, sopratutto, una maggiore stabilità rispetto alle variazioni di temperatura, occorrerà implementare un oscillatore a quarzo esterno.

La RAM

L' algoritmo che abbiamo visto richiede tre contatori, ovvero tre locazioni di RAM dati da utilizzare come contatori e che sono state chiamate con le label **`d1`**, **`d2`**, **`d3`** .

Sarà necessario definire questi contatori nel sorgente **prima** di poterli utilizzare.

"**Definire**" significa collegare i nomi (label) a locazioni della memoria.

Un modo molto semplice è quello di dichiarare delle eguaglianze (***equates***) tra le label e gli indirizzi assoluti della RAM.

EQU

EQU è una direttiva che abbiamo già considerato e di cui rivediamo le caratteristiche.
La prima cosa da notare è che la label inizia in prima colonna e con questo viene definita. Dal momento in cui l'Assembler incontra la linea, nella compilazione alla **label** sarà sostituito il valore indicato.
Quindi:

```
   base  EQU  0x25
```

fa si che la label **base** valga 25h. E' possibile anche una definizione usando il segno = :

```
  base = 0x25
```

Ma in tal caso, se ritratta di valori costanti, è possibile trattarli nella forma data dalla direttiva **CONSTANT**, che definisce una costante:

```
  CONSTANT  base = 0x25
```

Dal punto di vista formale, può essere preferibile utilizzare quest'ultima scrittura quando si definiscono delle equivalenze, giusto per differenziare le label relative a questo genere di oggetti dalle label auto definite (che iniziano in prima colonna), che sono label relative agli indirizzi delle linee di istruzione collegate.

L' uso di **#define** è altrettanto valido ed è solitamente relativo ad una sostituzione di testo:

```
#define  LED  PORTB,RB5
```

ma anche:

```
#define  base  0x25
```

EQU e CBLOCK per definire la RAM dati

in base quanto sopra, nel nostro caso, sappiamo che la RAM dati disponibile in **12F519** parte dall' indirizzo 07h; quindi, con la direttiva **=** assegniamo le tre label a tre successivi indirizzi di memoria:

```
;*****************************************************************
;                        MEMORIA RAM
; general purpose RAM
d1 = 0x07          ; contatori per ritardo
d2 = 0x08
d3 = 0x09
```

In questo modo dichiariamo che le tre label sono equivalenti per il compilatore ai tre indirizzi nell' area della RAM dati:

```
d1 = 7
d2 = 8
d3 = 9
```

Il modo è molto semplice e si trovano in giro fin troppi cattivi esempi di programmazione che lo utilizzano, ma **non è per niente consigliabile** a meno di avere la necessità, per qualche ragione, di conoscere o fissare con immediata certezza la corrispondenza tra label e indirizzo.
Anche la forma:

```
;*****************************************************************
;                       MEMORIA RAM
; general purpose RAM
d1   EQU   0x07          ; contatori per ritardo
d2   EQU   0x08
d3   EQU   0x09
```

ha valore del tutto analogo.

La motivazione del consiglio è semplice: **osservando le caratteristiche di diversi componenti, anche solo della famiglia Baseline, troviamo che:**

- **la quantità di RAM disponibile può essere diversa da chip a chip**
- **l' indirizzo di partenza dell' area dati è variabile**

L' equivalenza diretta con valori assoluti fa sì che il codice risultante sia solo minimamente portabile, in quanto per altri chip, con diverso inizio della RAM, occorrerà la ri scrittura di tutti gli equates. Ad esempio, se utilizzassimo il celebre **16F84A**, la sua RAM inizia a **0Ch** e non a 07h. Quindi il sorgente precedente andrebbe riscritto:

```
;*****************************************************************
;                       MEMORIA RAM
; general purpose RAM
d1   EQU   0x0C          ; contatori per ritardo
d2   EQU   0x0D
d3   EQU   0x0E
```

Se usassimo il **16F628**, la cui RAM inizia **20h**:

```
;*****************************************************************
;                       MEMORIA RAM
; general purpose RAM
d1   EQU   0x20          ; contatori per ritardo
d2   EQU   0x21
d3   EQU   0x22
```

Questo modo di dichiarare una equivalenza tra label e locazioni di RAM è scorretto e inefficiente.

Se le locazioni di RAM da dichiarare fossero molte di più, ci si costringerebbe alla ri scrittura di più righe ogni volta che gli indirizzi di inizio cambiano, col rischio di errori e in contrasto con il concetto base che è quello di scrivere un sorgente quanto possibile portabile tra i vari processori.

Esistono modi di scrittura diversi, che evitano questo problema; possiamo iniziare con quello più comune, ovvero con l' uso della direttiva dell' Assembler **CBLOCK**.

CBLOCK - ENDC

La direttiva **CBLOCK** ha lo scopo di definire un blocco di memoria RAM. Essa:

- **fa riferimento alla memoria RAM dati**
- non va posta in prima colonna
- deve essere chiusa da **ENDC**
- può comprendere più righe, composte dalle label e da una indicazione di quante locazioni occupano.

Da notare che le label definite nel blocco non iniziano in prima colonna, anche si si tratta di una definizione, dato che questa dipende dalla direttiva. La sintassi di **CBLOCK** è semplice:

```
 CBLOCK indirizzo                                  [;commento eventuale]
<label> [:<incremento>][,<label>][,<label>] [;commento eventuale]
 ENDC
```

Se non viene dichiarato un incremento, alla label è riservata una sola locazione. Però, se occorre, la label può essere assegnata ad un gruppo di più locazioni consecutive,dipendenti da essa.
Ad esempio:

```
  CBLOCK 0x07
  d1:3            ; riserva 3 bytes per i counters
  buffer:8        ; riserva 8 bytes per il buffer
  ka              ; riserva 1 byte per la variabile ka
  ENDC
```

Assegna:

label	locazioni	Indirizzi
d1	3	07, 08, 09
buffer	8	da 0Ah a 12h
ka	1	13h

Da notare che in questo caso l' equivalenza simbolica per le prime tre locazioni è:

Label	Indirizzo in RAM
d1	07h
d1+1	08h
d1+2	09h

ovvero solamente la prima ha un vero e proprio "nome", cioè **d1**, mentre le successive, se devono essere identificate singolarmente, usano la label della prima e l' indicazione **+n** a seconda della loro posizione. Come abbiamo già visto per la direttiva **MACRO** che va usata in coppia con la **ENDM** , anche nel caso di **CBLOCK** la direttiva è composta da una coppia: **esaurite le dichiarazioni, è indispensabile chiudere il blocco, con la direttiva ENDC**.

Se dimentichiamo questo particolare, il compilatore genererà errori, dato che non è in grado di definire dove il blocco viene terminato. Questi errori impediscono la conclusione corretta della compilazione e il file ***.hex*** non viene creato.

Nel file ***.err*** e nel file **.lst** generati dall' Assembler sono evidenziati gli eventuali errori della compilazione e la loro analisi permette di risolvere rapidamente il problema.

Quindi:

```
   CBLOCK 0x07      ; inizio area RAM
     d1,d2,d3       ; contatori per ritardo
   ENDC
```

L' oggetto di **CBLOCK** è l' indirizzo di memoria da cui partire, in questo caso 07h, dove inizia la RAM dati.

All' interno del blocco, dichiariamo le label delle locazioni di RAM da utilizzare; l' algoritmo scelto ne richiede tre, denominate **d1**, **d2** e **d3**. Potremo benissimo utilizzare altre label di nostro gradimento; quello che importa è comprendere il meccanismo: la direttiva **CBLOCK** indica al compilatore che nel blocco di label successive esiste questa relazione con indirizzi nella mappa della RAM dati. Nel nostro caso abbiamo:

Label	Indirizzo in RAM
d1	07h
d2	08h
d3	09h

Possiamo anche usare la forma:

```
   CBLOCK 0x07      ; inizio area RAM
     d1:3
   ENDC
```

E questo genera la seguente situazione:

Label	Indirizzo in RAM
d1	07h
d1+1	08h
d1+2	09h

Dato che la label è assegnata solo al primo valore, mentre i successivi sono da esso dipendenti. Così il sorgente andrebbe scritto:

```
; Delay = 1 secondo
; Clock frequency = 4 MHz
; Actual delay = 1 secondo = 1000000 cicli
Delay1s:                ; 999997 cicli
  movlw   0x08          ; inizializza counters
  movwf   d1
  movlw   0x2F
```

```
  movwf   d1+1
  movlw   0x03
  movwf   d1+2
Delay1s_0:                ; loop di conteggio -<-|
  decfsz  d1, f           ; d1=d1-1              |
   goto   $+2             ;->--|                 |
  decfsz  d1+1, f         ;    |  d2=d2-1        |
   goto   $+2             ;-<>-|                 |
  decfsz  d1+2, f         ;    |  d3=d3-1        |
   goto   Delay1s_0       ;-<>-|----->>------>>--|
; fine conteggio 999997 cicli - ora somma i 3 mancanti
  goto    $+1             ;->-|   3 cycles
  nop                     ;-<-|
```

Non c'è una regola particolare per usare uno o l' altro modo; anche assegnando label ad ogni locazione non si creano problemi al compilatore, che, comunque, lavora a velocità dipendenti essenzialmente dalle capacità del personal computer che lo supporta e che sono ampiamente abbondanti; di conseguenza l' aumento degli elementi da trattare non penalizza sensibilmente il tempo di compilazione.
Per contro, l' uso di una sola label per un gruppo di locazioni potrà essere vantaggioso quando il gruppo costituisce un unicum rispetto ad una certa funzione, ad esempio un buffer circolare, dove non ha alcuna importanza dare un nome proprio a ciascuno degli elementi che lo compongono.

Come nel caso degli equates, occorre conoscere dove allocare il blocco della RAM dati, ma, rispetto a quella modalità, dove, cambiando indirizzo di inizio RAM occorre riscrivere ogni singolo equate, qui basterà cambiare unicamente l' indirizzo di inizio blocco.

Così, per i casi visti sopra, ora **basta modificare il sorgente in un solo punto**:

```
;16F84A
   CBLOCK 0x0C      ; inizio area RAM
     d1,d2,d3       ; contatori per ritardo
   ENDC
```

e :

```
;16F628
  CBLOCK 0x20       ; inizio area RAM
     d1,d2,d3       ; contatori per ritardo
   ENDC
```

Una sensibile semplificazione e senza alcun appesantimento del sorgente !

CBLOCK vs. ORG

 Ricordiamo che quando si parla di "indirizzi", nei PIC, realizzati in base all' architettura Harvard, ci troviamo di fronte a due distinte aree di indirizzi:

- gli indirizzi della memoria programma (Flash)
- gli indirizzi della memoria RAM, che comprende anche gli SFR

Quindi, esistono, ad esempio, un indirizzo 00h nella memoria programma e un indirizzo 00h nella memoria RAM.

Le due locazioni sono ben differenti e, anche se hanno lo stesso indirizzo, si trovano su due bus completamente diversi: il primo corrisponderà ad una locazione della memoria Flash detta vettore di reset, dove si troverà la prima istruzione del programma; il secondo corrisponderà ad una locazione della RAM in cui solitamente si trova un registro SFR.

Dai fogli dati di ogni componente possiamo consultare la mappa della memoria (e sarebbe meglio dire "memorie") e le relative caratteristiche.

Occorre avere ben presente questa particolarità, in quanto l' Assembly permette di agire sia su un bus che sull' altro, ma si tratta di cose ben diverse.

A livello di Assembler, notiamo che esistono due comandi specifici per posizionarsi all' inizio di un blocco di memoria:

- **CBLOCK che fa riferimento all' area RAM**, dove determina l' inizio di un blocco di definizioni che viene delimitato alla fine da **ENDC**
- **ORG fa riferimento alla memoria programma** e determina a quale indirizzo si posiziona l' istruzione che sarà successivamente indicata. Non richiede, quindi, una "chiusura".

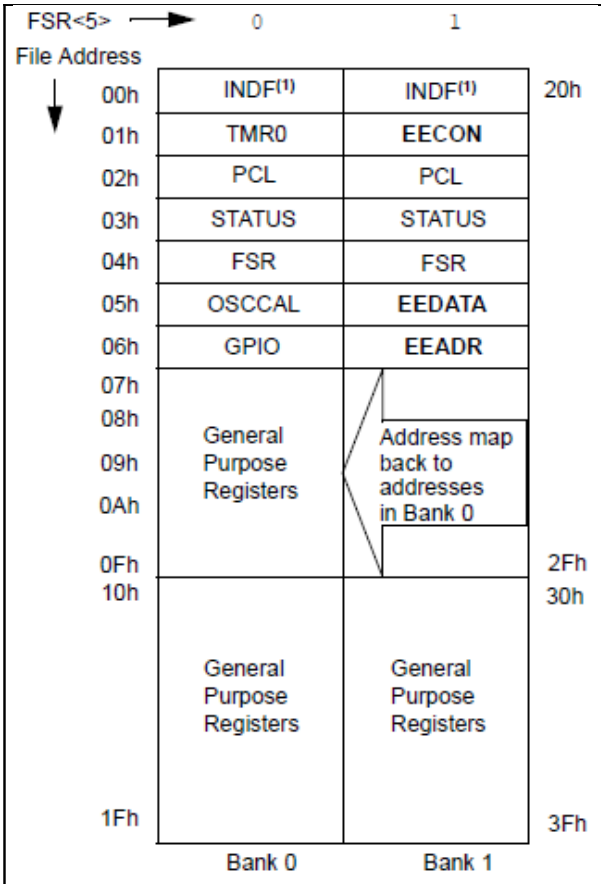
Ripetiamo: le due cose sono ben diverse e non vanno confuse. Occorre quindi avere chiara la situazione della memoria nei chip.

RAM e MEMORIA PROGRAMMA

Esistono di diverse aree di memoria:

- Memoria RAM : contiene i registri SFR e la RAM dati. Essendo una RAM, il contenuto va perso alla mancanza di tensione di alimentazione.
- Memoria programma: in tecnologia Flash, contiene il programma. Il contenuto permene inalterato fno a che viene modificato da una ri scrittura del chip.
- Memoria EEPROM: non presente in tutti i chip, costituisce una area di memoria cancellabile e scrivibile elettricamente per conservare dati anche in mancanza di tensione

Ecco ad esempio la mappa dell’ area Ram o memoria dati per 12F519:



Questa area di memoria è divisa in due banchi, di cui il primo inizia all' indirizzo 00h e il secondo a 20h.

I primi 7 bytes di ogni banco sono occupati da registri SFR (quelli che controllano le periferiche integrate).

Ad esempio, all' indirizzo 03h troviamo il registro di STATUS, all' indirizzo 02h troveremo il PCL (Program Counter Low).

A questi fa seguito un' area di General Purpose RAM, ovvero della vera e propria RAM dati.

Dunque, per questo processore la memoria RAM utilizzabile inizierà all' indirizzo 07h.

I bytes da 07h a 0fh sono accessibili da entrambi i banchi, essendo uno copia dell' altro.

I bytes da 10h a 1Fh sono accessibili solo sul banco 0 e quelli da 30h a 3Fh solo dal banco 1.

Altri processori avranno differenti mappe di memoria RAM, con meno o più banchi e con registri SFR in quantità proporzionale al numero delle periferiche e delle funzioni integrate. Si va da 1 a 32 banchi e da 7 a molte decine di SFR.
La dimensione della RAM dati varia tra un paio di decine di bytes e vari kB.

Il contenuto della RAM è di tipo volatile, ovvero quanto scritto va perso al momento della mancanza della tensione di alimentazione. Le singole celle possono conservare, finchè alimentate, un dato binario 1 o 0 che vi è stato scritto (1 bit).

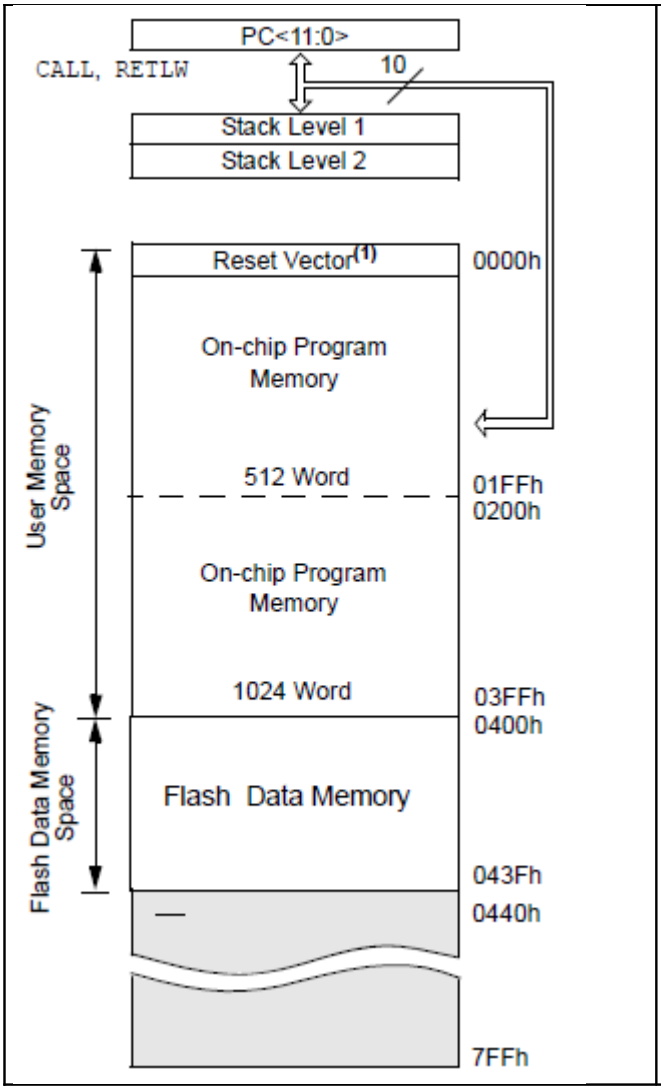
Quindi dati in RAM e programmazione degli SFR non sono permanenti.

In ogni caso, la direttiva **CBLOCK** / **ENDC** fa riferimento a questa area.

In particolare, dobbiamo notare che:

- l' ampiezza di una locazione di RAM per i Baseline è pari a 8 bit (1 byte).
  Per questa ragione si parla di processori a 8 bit, intendendo la massima ampiezza del dato che possono trattare.
- Le istruzioni che muovono dati permettono l' accesso a questa area
- Gli SFR, anche se scrivibili e leggibili come locazioni di RAM dati, non sono previsti per questo scopo, ma servono a gestire periferiche e funzioni e non vanno utilizzati in altro modo (salvo eccezioni possibili).

Per contro, le locazioni di **memoria programma** hanno una mappa differente, essendo accessibili da un diverso bus:



Anche qui la memoria inizia con l' indirizzo 000h, ma non ha nulla a che fare con la precedente. Qui siamo su un bus diverso, il bus istruzioni, che ha ampiezza 12 bit nei Baseline.

Di conseguenza una locazione di memoria programma sarà ampia 12 bit.

In particolare, la locazione iniziale all' indirizzo 0000h è definita come Vettore del Reset, ovvero contiene la prima istruzione da eseguire al momento dell' accensione (Power On Reset o POR) o dopo un reset dovuto a MCLR o altre cause.

Nei PIC Midrange e successivi esiste anche un' altra locazione tipica, ovvero il Vettore di Interrupt, che nei Baseline non è presente, mancando questa funzione.

La memoria, in tecnologia Flash, ha una ampiezza che va da 512 "bytes" in su (ricordiamo che qui il "byte" vale 12 bit); quindi meglio definirle come word.

Il Program Counter (PC) è il registro che tiene conto della posizione corrente nella lista di esecuzione delle istruzioni, coadiuvato dallo stack.

Occorre avere ben presente che il contenuto della memoria programma viene scritto durante la fase di programmazione del chip e contiene il "programma" o firmware. Il contenuto è conservato anche in mancanza di tensione, per periodi molto lunghi (microchip garantisce almeno 100 anni…).
Per poter cambiare questo contenuto occorre una ri programmazione.

Nei Baseline il contenuto della memoria programma non è accessibili dal programma stesso. Questa azione diventa possibile nei PIC maggiori..

Errore!

Una avvertenza importante:

L' Assembler, durante la compilazione, esegue quanto abbiamo indicato nel sorgente. E' in grado di controllare e segnalare errori di sintassi e di forma e, in alcuni rari casi, introduce automaticamente piccole correzioni formali.

Ma non è assolutamente in grado di correggere errori di logica !

Ad esempio, se indichiamo:

```
CBLOCK 0x0C        ; inizio area RAM
   d1,d2,d3        ; contatori per ritardo
 ENDC
```

e lo riferiamo ad un **PIC16F628**, il compilatore non indicherà alcun errore, dato che esiste memoria RAM a questi indirizzi. Ma non si tratta di RAM dati, bensì di registri SFR che controllano delle periferiche ! Ricordiamo che la memoria RAM utilizzabile come memoria dati e i registri SFR sono contigui e si trovano nella stessa area.

Nel caso in esempio avremmo definito come RAM tre locazioni che corrispondono rispettivamente a **PIR1**, una locazione non implementata e **TMR1L**; ci si può immaginare le possibili conseguenze sull' esecuzione del programma che cerca di utilizzare come memoria dati alcuni registri di controllo delle periferiche o locazioni non implementate in cui scrittura e lettura non sono possibili!

Fortunatamente, nel secondo passo della compilazione, l' Assembler confronterà i dati forniti con i limiti imposti dai file ***.inc*** e ***.lkr*** e segnalerà, alla prima istruzione che tenta di utilizzare questa "RAM" erroneamente definita, un messaggio del genere:

```
Warning[219]: Invalid RAM location specified.
```

Attenzione, però, perchè si tratta di ***Warning***, ovvero di avvisi che NON IMPEDISCONO la compilazione dell' oggetto, per cui si avrà nella finestra di Output il rassicurante:

```
BUILD SUCCEEDED
```

e di conseguenza un file ***.hex*** sarà generato, ma evidentemente non farà quello che ci si aspetta !!!

Il comportamento del compilatore è perfettamente corretto: la direttiva **CBLOCK** dispone la definizione di un blocco in RAM. Potremmo, per qualche ragione, voler applicare questo blocco a registri SFR e non a RAM dati: il compilatore non può conoscere le nostre intenzioni e non può prendere decisioni autonome. Quindi compila. Però, i creatori del programma hanno previsto una distrazione del programmatore ed hanno introdotto dei controlli di congruenza; di conseguenza il compilatore genera messaggi di avvertimento per il programmatore in modo che possa correggere eventuali sbagli. Ma lascia, come è logico, all' utente il peso delle sue decisioni.
Ovvero, il compilatore fa quello che gli dite di fare: se si tratta di errori formali che gli impediscono di interpretare le richieste, la compilazione non andrà a buon fine. Ma se avete richiesto di eseguire una azione e in realtà ne era necessaria un' altra, questo nessun compilatore è certo in grado di correggere, che sia Assembler o C o Basic.

Avvertenza importante.

Ribadiamo che, se si omette di leggere l' intero report di compilazione o, peggio, si sono inserite linee di soppressione dei messaggi, si rischia di ottenere un programma non funzionante senza riuscire a capirne le motivazioni.

Da qui il consiglio di **non inserire mai righe di comando per l' abolizione di messaggi del compilatore, a meno che siate perfettamente sicuri delle possibili conseguenze**.
Purtroppo questa stupida pratica è estremamente comune negli esempi presenti sul WEB e, nelle mani del principiante, è un poderoso mezzo per rendersi difficile la stesura del sorgente ed il suo debug.

Le MACRO

Dobbiamo ora accendere e spegnere il LED. Abbiamo visto in precedenza la creazione di una macro istruzione (es. 1Aw). Possiamo fare un ripasso, rivedendo i punti principali.

Una MACRO (abbreviazione per **macro istruzione**) **è un tratto di codice che viene raccolto sotto una singola label. Ogni volta che il compilatore incontra questa label, la sostituisce con il testo sottinteso.**

Si può pensare ad una analogia con il già visto `#include`, ma si tratta di due cose molto differenti:

- `#include` è un comando del linguaggio che permette al compilatore di includere nel testo sorgente un altro testo che si trova al di fuori, come è il caso di una libreria
- il comando `MACRO` è pure una funzione dell' Assembler (che si chiama. per la precisione, Macro Assembler), ma ha lo scopo di sostituire una serie di istruzioni con una singola label

La macro fa sempre parte della filosofia base dell' Assembly per ottenere sorgenti costituiti quanto possibile da simboli, che, se usati con accortezza, li rendono semplici e immediati da leggere, cancellando una delle asperità di un linguaggio così essenziale.

La creazione di simboli, inoltre, consente di ottenere due vantaggi sensibili:

- permette di scrivere una volta sola nel sorgente una sequenza di istruzioni che devono essere ripetute più volte, con vantaggio di una maggiore efficienza nella stesura del testo
- consente di avere una label sola che, se correttamente denominata, permette rilettura e comprensione maggiore del listato.

In questo senso è possibile scrive macro di uso generale o specifiche per la soluzione di un problema e collezionarle in librerie che sarà possibile includere nel sorgente con la direttiva `#include` .

Nel nostro sorgente possiamo scrivere, allora:

```
;*****************************************************************
;                            LOCAL MACROS
; Comandi per il LED
LED_ON  MACRO
        bsf   LED
          ENDM
LED_OFF MACRO
        bcf   LED
          ENDM
```

La label obbligatoria identifica la macro. Il codice, in questo caso, è costituito da una sola riga:

- `bsf LED` che accende il LED, per la macro **LED_ON**
- `bcf LED` che spegne il LED, per la macro **LED_OFF**

Questo significa che:

- ogni volta in cui scriveremo la label **LED_ON** nel sorgente, il compilatore la sostituirà con la linea `bsf LED`.

L' utilità è evidente: nel sorgente, al posto di un non chiarissimo **bsf LED** ci troveremo a leggere un più evidente **LED_ON**. Ovviamente se la dizione non è di vostro gusto, potete cambiare nel sorgente la label, ad esempio con:

```
;*****************************************************************
;                           LOCAL MACROS
; Comandi per il LED
ACCENDI_LED  MACRO
      bsf   LED
            ENDM
SPEGNI_LED  MACRO
      bcf   LED
            ENDM
```

o qualsiasi altro nome valido. L'importante è che le label si attengano alle regole dell' Assembler.

E' anche estremamente utile che **siano quanto possibile degli mnemonici in grado di collaborare alla comprensione della logica del programma**.
Inoltre, può essere opportuno utilizzare lettere maiuscole per le macro, per distinguerle da altri elementi del sorgente.

In alcuni esempi, i programmatori di Microchip utilizzano l' indicazione **m** per discriminare una macro; così si avrebbe **mACCENDI_LED**; si tratta comunque di scelte "personali" che hanno sempre e solo lo scopo finale di rendere più facilmente comprensibile quanto si sta scrivendo.

Quale è la reale utilità dell' uso delle macro in un caso come questo?

Se ci limitiamo al programma in esempio, così semplice, si potrebbe dire nessuno. Ma se pensiamo di avere di fronte situazioni più complesse, le ragioni per l' uso delle macro diventano evidenti.
Abbiamo, ad esempio, la necessità di azionare più LED, collegati ai pin **GP0**, **GP1**, **GP2**, **GP5**.
Se usassimo una scrittura a "basso livello" avremmo necessità di una istruzione **bsf** per accendere un LED e di una istruzione **bcf** per spegnerlo; queste istruzioni dovrebbero avere come oggetto il registro GPIO e il bit da azionare, creando un sorgente un po' pesante:

```
; comando LED
      bsf   GPIO, GP5   ; accendi LED allarme generale
... altre istruzioni
      bsf   GPIO, GP2   ; accendi LED sovratemperatura
      bcf   GPIO, GP0   ; spegni LED ok
... ecc
```

che potrebbe essere poco chiara. Se però definiamo:

```
#define Led_allarme   GPIO, GP5
#define Led_overtemp  GPIO, GP2
#define Led_ok        GPIO, GP0

Allarme_on   MACRO
```

```
             bsf Led_allarme
             ENDM
Overtemp_on  MACRO
             bsf Led_overtemp
             ENDM
Ok_no        MACRO
             bcf Led_ok
             ENDM
```

la lista precedente diventerà:

```
; comando LED
Allarme_on
... altre istruzioni
Overtemp_on
Ok_no
... ecc
```

che non hanno neppure bisogno di commenti.

Inoltre, un reale vantaggio dato dai simboli consiste nel fatto che se dobbiamo scambiare le funzioni sugli I/O, basterà un unico cambio nelle dichiarazioni delle label e basta. Ad esempio:

```
#define Led_allarme   GPIO, GP0
#define Led_overtemp  GPIO, GP1
#define Led_ok        GPIO, GP5
```

Non occorre altra modifica al sorgente, in quanto avendo dichiarato diversamente le label, automaticamente ogni volta che essere saranno richiamate si avrà la giusta assegnazione.

Anche i simboli-macro hanno la stessa prestazione. Se l' azione di allarme della sovra temperatura richiede anche di azionare un cicalino, basterà cambiare solamente la macro relativa:

```
#define Cicalino GPIO, GP2

Overtemp_on  MACRO
             bsf Led_overtemp
             nop
             nop
             bsf Cicalino
             ENDM
```

mentre il resto del sorgente resterà invariato.

Dovrebbe iniziare ad essere ben chiaro l' enorme vantaggio dato dall' uso dei simboli.

Il loop di lampeggio

La programmazione del pin come direzione e come funzione uscita è del tutto identica a quella vista nell' esercizio precedente.

Dobbiamo inserire il loop di tempo per il lampeggio.

 Con il termine ***loop*** (anello, in inglese) si intende una serie di istruzioni che continua ciclicamente a ripetersi. Si potrà avere un loop indefinito, quando non c'è un punto di uscita; oppure un loop condizionato, il cui termine è stabilito da specifiche condizioni.

Qui si tratta di un loop del primo tipo, dato che il lampeggio deve proseguire indefinitamente, fino a che c'è tensione di alimentazione.

Inseriamo quindi il loop di lampeggio secondo l' algoritmo visto prima:

```
mainloop:             ;<<-----<<<-------<<----|
; accende LED         ;                       |
  LED_ON              ;                       |
                      ;                       |
Delay1s:              ; attesa 1 s - 999997 cicli
  movlw   0x08        ; inizializza counter   |
  movwf   d1          ;                       |
  movlw   0x2F        ;                       |
  movwf   d2          ;                       |
  movlw   0x03        ;                       |
  movwf   d3          ;                       |
Delay1s_0             ; loop di conteggio     |
  decfsz  d1, f       ; d1=d1-1               |
   goto   $+2         ;                       |
  decfsz  d2, f       ; d2=d2-2               |
   goto   $+2         ;                       |
  decfsz  d3, f       ; d3=d3-1               |
   goto   Delay1s_0   ;                       |
; fine 999997 cicli - ora somma i 3 mancanti |
  goto    $+1         ; 3 cicli               |
  nop                 ;                       |
                      ;                       |
; spegne LED                                 loop
  LED_OFF             ;                       |
                      ;                       |
Delay1s:              ; attesa 1 s - 999997 cicli
  movlw   0x08        ; inizializza counter   |
  movwf   d1          ;                       |
  movlw   0x2F        ;                       |
  movwf   d2          ;                       |
  movlw   0x03        ;                       |
  movwf   d3          ;                       |
Delay1s_0             ; loop di conteggio     |
  decfsz  d1, f       ; d1=d1-1               |
   goto   $+2         ;                       |
  decfsz  d2, f       ; d2=d2-2               |
```

```
   goto    $+2         ;                          |
  decfsz   d3, f       ; d3=d3-1                  |
   goto    Delay1sa_0  ;                          |
; fine 999997 cicli - ora somma i 3 mancanti      |
  goto     $+1         ; 3 cicli                  |
  nop                  ;                          |
                       ;                          |
; loop                                            |
  goto     mainloop    ;>>------>>>-------->>---|
```

Questa volta il programma è un loop, ovvero una esecuzione continua di un determinato gruppo di istruzioni.

BUILD FAILED !

Proviamo a compilare il sorgente (**2Ae_519.asm**). Sorpresa: otteniamo **alcune indicazioni di errore** !!!

```
Error[116] C:\PIC\A&C\CORSOA\BASELINE\A&C_2A\2A_519.ASM 176 : Address label duplicated or different in second pass (Delay1s)
Error[116] C:\PIC\A&C\CORSOA\BASELINE\A&C_2A\2A_519.ASM 183 : Address label duplicated or different in second pass (Delay1s_0)
Halting build on first failure as requested.
-------------------------------------------------------------------
Debug build of project `C:\Documents and Settings\afg\Desktop\SUPERCORSO\prove_corso\2aerr.mcp' failed.
Language tool versions: MPASMWIN.exe v5.43, mplink.exe v4.41, mplib.exe v4.41
Preprocessor symbol `__DEBUG' is defined.Wed Jul 03 13:52:03 2013
-------------------------------------------------------------------
BUILD FAILED
```

Cosa significa?
Il messaggio **BUILD FAILED** indica che la compilazione non è stata portata a termine e non è stato generato il file ***.hex*** a causa di qualche grave errore.
All' inizio del messaggio troviamo la descrizione di questo errore:

Error[116] C:\PIC\A&C\CORSOA\BASELINE\A&C_2A\2A_519.ASM 176 : Address label duplicated or different in second pass (Delay1s)
Error[116] C:\PIC\A&C\CORSOA\BASELINE\A&C_2A\2A_519.ASM 183 : Address label duplicated or different in second pass (Delay1s_0)

Si tratta, come indicato, della duplicazione di label (**`error [116]`**), coè, in questo caso, due label che sono state definite ad un certo punto del sorgente, ma che vengono ri definite successivamente. Questo non è ammissibile in quanto la stessa label non può corrispondere a valori diversi

La label a inizio riga

Occorre ancora precisare che:

Quando poniamo una label all' inizio della riga, in modo che essa inizi in prima colonna, indichiamo al compilatore che quella label deve essere definita e il suo valore sarà quello dell' indirizzo di memoria programma a cui verrà posta l' istruzione successiva.

Da quel momento alla label il compilatore sostituirà l' indirizzo assoluto durante la creazione del file eseguibile.

Quindi, è evidente che NON è possibile avere nello stesso sorgente una stessa label che viene dichiarata in questo modo in due punti diversi del testo: questo vorrebbe dire che la label ha un valore x la prima volta che appare nel sorgente ed un valore y la seconda volta. Ovviamente il compilatore non può accettare questa situazione e genera un errore, terminando le operazioni.

Nel nostro caso, le label **`Delay1s`** e **`Delay1s_0`** si trovano nel sorgente ripetute due volte in due posizioni diverse; il tentativo di far corrispondere label con lo stesso nome a locazioni della memoria programma differenti è riconosciuto dal compilatore, che, non avendo modo di correggere la situazione, interrompe la compilazione con la segnalazione di errore.

Una nota ulteriore di chiarimento, per chi può avere qualche dubbio:

Una cosa è la definizione della label, ovvero l' operazione di far corrispondere un valore assoluto alla label stessa, attraverso la manovra di posizionarla in prima colonna.
Ben diverso è il fatto di richiamare la label come oggetto di una operazione.

Nel primo caso la definizione della label non può essere ripetuta per evidenti ragioni logiche : se dichiaro che la label **`Target`** vale 57h e poi più avanti nel testo dichiaro nuovamente che **`Target`** vale 11h, le due cose sono in conflitto e il compilatore si arresta.
Per essere ancora più chiari: se in un punto ho definito un valore per la label, ad esempio 57h, il compilatore ogni volta che la incontra la sostituisce con questo valore.
Ma se ad un certo punto cerco di definire nuovamente la label con il valore 11h, il compilatore si trova davanti alla situazione 57h = 11h, il che è assurdo.

Posso definire **la label una sola volta, dopo di che, a meno di cancellarne la definizione, essa manterrà il suo valore inalterato**.

Se, invece, una volta dichiarato un certo valore per **`Target`** uso la label come oggetto di una istruzione, ad esempio **`goto Target`**, la cosa è perfettamente logica, in quanto indico al compilatore non di definire nuovamente la label, ma di usarla per il suo valore equivalente.
Questo posso farlo **un numero illimitato di volte nello stesso testo sorgente**.

ERROR vs. WARNING vs. MESSAGE

Una nota su errori (**Error**) e avvertimenti (**Warning** e **Message**): nel caso in esame osserviamo che qui il compilatore ha inviato un messaggio di **Error** e non di **Warning** o **Message**.

Per il compilatore **MPASM** c'è differenza tra i due messaggi:

- **Error** corrisponde ad una situazione che il compilatore non può correggere e che determina il termine della compilazione senza generare il file .hex. Fino a che persiste un **Error** la compilazione non potrà essere conclusa.
- **Warning** , come visto prima, corrisponde ad una situazione di possibile errore, non correggibile dal compilatore, o di piccolo errore formale corretto dal compilatore. La compilazione va a buon fine, ma l' utente viene avvertito del possibile problema in modo tale da poter intraprendere azioni correttive, se necessarie.
- **Message** è analogo a Warning, ma meno stringente

Queste segnalazioni sono codificate con numeri che ne identificano le caratteristiche e indicazione della riga del listato in cui sono applicate.

Sono elencati nel file ***.err*** e presenti nel file ***.lst***, oltre che visibili alla fine della compilazione nella finestra ***Output***, cartella ***Build***.

Ad esempio:

```
Error[151] C:\Pic\pro.asm 17 : Operand contains unresolvable labels or is too complex
```

Indica che nella riga **17** del listato relativo al file **C:\Pic\pro.asm** è stato rilevato un errore codificato come **151**, il che corrisponde all' uso di un operando costituito da una funzione non risolvibile o troppo complessa.

```
Warning[220] C:\Pic\pro1.asm 158 : Address exceeds maximum range for this processor.
```

Indica che alla riga **158** del listato derivato dal file **C:\Pic\pro1.asm** è stata rilevato un indirizzo che supera il limite di memoria del processore dichiarato.
E così via. Questi messaggi possono anche essere generati dall' utente con l' uso di due direttive dell' Assembler, allo scopo di servire da avviso per situazioni di compilazione non ammesse.

Troviamo tutti i dettagli di questo argomento nel documento MPASM User's Guide di Microchip

Correggiamo l' errore…

Per correggere la situazione occorre modificare il sorgente, attribuendo label diverse. Ad esempio, cambiandole leggermente nel secondo loop di tempo.

```
; spegne LED                                  loop
  LED_OFF             ;                       |
                      ;                       |
Delay1sa:             ; attesa 1 s - 999997 cicli
; attesa 1 s - 999997 cicli                   |
  movlw   0x08        ; inizializza counter   |
  movwf   d1          ;                       |
  movlw   0x2F        ;                       |
  movwf   d2          ;                       |
  movlw   0x03        ;                       |
  movwf   d3          ;                       |
Delay1sa_0            ; loop di conteggio     |
  decfsz  d1, f       ; d1=d1-1               |
   goto   $+2         ;                       |
  decfsz  d2, f       ; d2=d2-2               |
   goto   $+2         ;                       |
  decfsz  d3, f       ; d3=d3-1               |
   goto   Delay1sa_0  ;                       |
; fine 999997 cicli - ora somma i 3 mancanti  |
  goto    $+1         ; 3 cicli               |
  nop                 ;                       |
                      ;                       |
; loop                                        |
  goto    mainloop    ;>>------>>>-------->>---|
```

Quindi, il primo blocco di ritardo ha la label del suo loop chiamata **`Delay1s_0`**, ma il secondo ha una label differente, **`Delay1sa_0`** , anche se il loop esegue la stessa identica operazione.

Ovvero, se modifichiamo le seconde due label come indicato, il problema è superato.

Per fare questo, possiamo modificare con l' editor il file **2Ae_519.asm**, il che si effettua semplicemente correggendo il testo con l' editor di MPLAB. Il file corretto è salvato automaticamente con lo stesso nome.

Però si tratta di una pratica del tutto da scartare: si può immaginare quali siano i problemi relativi all' obbligo di non duplicare label in una situazione in cui siano presenti numerose routines e le procedure vengano richiamate in molti punti del programma: ogni volta dovremmo aggiornare manualmente label nel sorgente per uno stesso **blocco logico** di istruzioni. Insensato!!!

Possiamo fare molto meglio.

Le Subroutine

La parola chiave della soluzione è "**blocco logico**".

Se nel nostro programma utilizziamo una stessa routine di tempo più volte è del tutto inutile e poco sensato ripeterla nel sorgente ogni volta che sia necessaria. La pratica corretta è trasformarla in un blocco logico, ridotto ad una singola label, e richiamare questa. Un poco come abbiamo già visto per le MACRO.

Ripetiamo le regole generali di un buon sorgente per un microcontroller, che sono:

- **scrivere un sorgente facilmente comprensibile e di facilme manutenzione**
- **rendere quanto possibile modulari le strutture per permetterne il ri utilizzo**
- **utilizzare il minimo possibile delle risorse**

Per ottenere questo, introduciamo la **subroutine**.

Una SUBROUTINE è un tratto di codice che viene raccolto sotto una singola label ed è terminato dall' istruzione `return`.
L' accesso a questo codice si effettua con l' istruzione `call`.

La funzione della subroutine è quella di costituire una parte di codice a cui il programma viene deviato dalla chiamata a subroutine (`call`) ed eseguito il quale il Program Counter viene riportato alla riga successiva a quella della chiamata con una istruzione ri rientro da subroutine (`return`).

Questa deviazione della linea di esecuzione si basa sugli automatismi che le due istruzioni sottintendo e che movimentano il valore contenuto nel Program Counter, facendo uso dello stack.
Durante le esercitazioni avremo modo di ampliare e definire in dettaglio questi elementi.

CALL

Incontriamo qui una nuova istruzione: `call` . il suo scopo è quello di “chiamare” (***to call***) una subroutine, in questo senso:

- `call` modifica il ***Program Counter***

La sintassi è la solita ed un oggetto è obbligatorio.

[Label]	sp	call	sp	oggetto	sp	[;commento]

Questo oggetto corrisponde ad un indirizzo della memoria programma, indicato da un valore assoluto o da una label.
Ricordiamo la funzione del ***Program Counter***: per semplificare al massimo, esso è il registro che tiene conto degli indirizzi di memoria programma che dovranno essere eseguiti immediatamente dopo l’ istruzione corrente.
Normalmente il ***Program Counter*** punta alla riga successiva del sorgente, ma, nel caso della chiamata a subroutine, scatta il seguente meccanismo:

- l' indirizzo contenuto nel ***Program Counter*** viene copiato nello stack, speciale area di memoria RAM che è adibita esclusivamente a questa funzione
- nel PC viene copiato un indirizzo diverso, per la precisione quello indicato dall' oggetto dell' istruzione **`call`**

In questo modo, la prossima istruzione da eseguire sarà non quella della riga successiva, ma si "salterà" alle istruzioni nell' area indicata dall' oggetto.

In questo modo posso sospendere una sequenza di operazioni (main o programma principale) per passare ad eseguire altre istruzioni.

Alla fine del blocco logico delle istruzioni della subroutine, dobbiamo inserire l' opcode **`return`**, che effettua la seguente operazione:

- il contenuto del ***Program Counter***, che punterebbe all' istruzione successiva al **`return`**, viene modificato, recuperando dallo stack quello precedentemente salvato

Questo indirizzo è quello della istruzione successiva a **`call`**: questo fa sì che il flusso dell' esecuzione venga riportato al punto in cui era stato deviato.

RETURN

`return` non richiede alcun oggetto e la sua sintassi è:

[Label]	sp	return	sp	[;commento]

La funzione dell' opcode è quella di prelevare dalla cima (top) dello stack l' indirizzo depostovi dalla **`call`** e ripristinarlo nel ***Program Counter*** , in modo da riprendere l' esecuzione del programma nel punto in cui era stata sospesa dall' esecuzione della subroutine.

Nei Baseline l'istruzione non esiste ed è sostituita da:

RETLW

Il suo scopo è identico a quello di **`return`**, cioè chiudere l' esecuzione della subroutine, facendo rientrare nel flusso di istruzioni che era stato abbandonato dalla di chiamata della subroutine stessa. La sintassi è:

[Label]	sp	return	sp	oggetto	sp	[;commento]

E' obbligatorio un oggetto, che sarà un valore compreso tra 0 e 255 (00h-FFh).
Questo numero viene caricato nel registro W.

RETURN vs. RETLW

Chi ha provato ad utilizzare l' applet indicata per la creazione di routine di tempo, avrà notato che esse sono chiuse dall' istruzione **`return`** . Pechè qui usiamo invece l' istruzione **`retlw`** ?
Semplicemente perchè l' opcode **`return`** non esiste nel set dei Baseline e va sostituito con **`retlw`** la cui funzione è analoga:

- **`return`** sostituisce il contenuto del ***PC*** con quello del top dello stack
- **`retlw`** sostituisce il contenuto del ***PC*** con quello del top dello stack e, **in più**, carica nel registro **W** il valore indicato in oggetto.

Nel nostro caso, questo valore viene posto a 0, ma potrebbe essere qualsiasi tra 0 e 255 (FFh), dato che qui non viene considerato. Però, in altri casi, come ad esempio nelle lookup tables, come vedremo più avanti, **`retlw x`** ha una importanza fondamentale.

In sostanza, le due istruzioni sono completamente identiche, con la sola differenza che :

- **`return`** non richiede alcun oggetto
- **`retlw`** richiede un oggetto costituito da un valore numerico tra 00h e FFh, valore che sarà caricato in nel registro **W**.

Chi volesse verificare uno dei piccoli esempi di correzione automatica offerti dall' Assembler, può sostituire la linea **`retlw 0`** con un **`return`** e compilare.

Otterrà un messaggio:

```
Warning[227] : Substituting RETLW 0 for RETURN pseudo-op
```

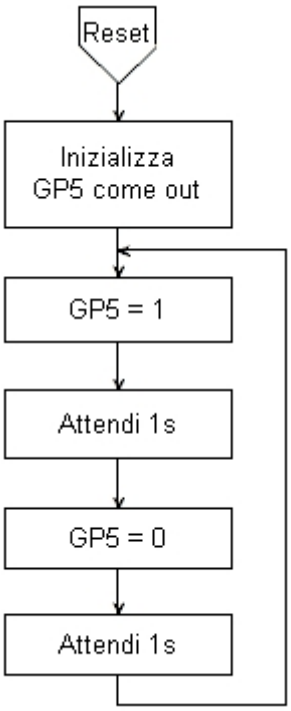
I creatori del compilatore, tenuta presente la "stranezza" della mancanza dell' istruzione **`return`** nei Baseline, hanno introdotto questa auto correzione, che, comunque, viene segnalata.
Essendo un ***Warning***, non blocca la creazione del file ***.hex***, ma avvisa l' utente che l' Assembler ha trovato qualcosa di non perfettamente a posto ed ha messo in atto una azione correttiva; sta all' utente accertarsi delle conseguenze della correzione.

Di nuovo, è importante leggere l' intero messaggio di compilazione oppure il file ***.lst*** per verificare quali sono state le azioni di MPASM.

Creare la subroutine

Per la creazione di una subroutine occorre raccogliere sotto un' unica label le istruzioni che la compongono; nel nostro caso, quelle che generano il ritardo.

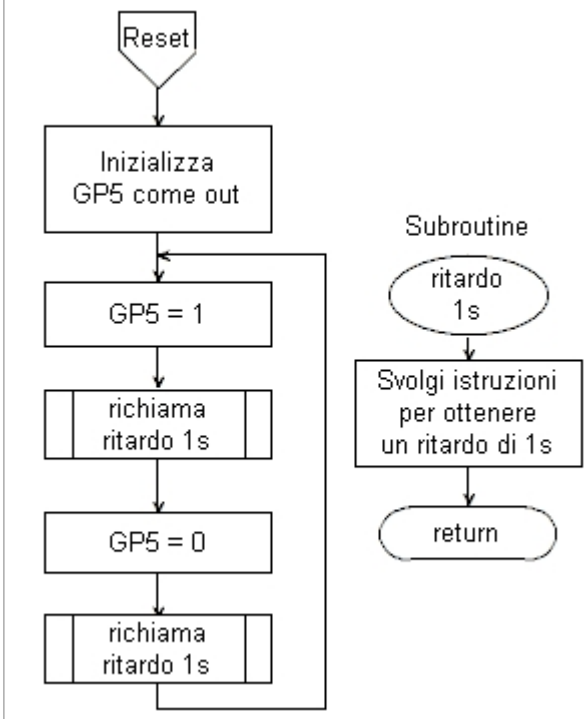
Lo vediamo meglio attraverso il flowchart.



Il diagramma di sinistra riporta la sequenza delle operazioni da eseguire.

A destra vediamo come si modifica con la creazione di una subroutine.
Il blocco logico di istruzioni che effettua il ritardo di 1 secondo non viene ripetuto due volte nel sorgente, ma viene scritto a parte e richiamato come subroutine.

Troviamo il sorgente in ***2As_519.asm*** , che, come al solito, stampiamo ed andiamo a commentare.



La struttura generale è la stessa della precedente versione e il loop di lampeggio sarà modificato così:

```
mainloop:             ;<---<<---|
; accende LED         ;         |
  LED_ON              ;         |
                      ;         |
; attesa 1s                      |
  call    Delay1s     ;         |
                      ;         |
; spegne LED                   loop
  LED_OFF             ;         |
                      ;         |
; attesa 1s                      |
  call    Delay1s     ;         |
                      ;         |
; loop                ;         |
  goto    mainloop    ;>>-->>---|
```

Ora il mainloop è molto semplificato e si legge senza difficoltà. Al posto della ingombrante sequenza di istruzioni del ritardo, una semplice chiamata a subroutine.
Il blocco di istruzioni del ritardo, da parte sua, viene leggermente modificato per trasformarsi in subroutine.

L' applet indicata consente di ottenere anche questa modifica in modo completamente automatico:

```
;#######################################################################
;                        SUBROUTINES
; Delay = 1 secondo @ Clock = 4 MHz
; Fosc/4 = 1 us , 1 secondo = 1000000 cicli
Delay1s:              ; 999990 cicli
  movlw   0x07        ; inizializza counters
```

```
  movwf   d1
  movlw   0x2F
  movwf   d2
  movlw   0x03
  movwf   d3
Dly1s_0:
  decfsz  d1, f
   goto   $+2
  decfsz  d2, f
   goto   $+2
  decfsz  d3, f
   goto   Dly1s_0
; fine conteggio 999990 cicli - ora somma i 10 mancanti
  goto    $+1       ; 6 cicli
  goto    $+1
  goto    $+1
  retlw   0         ; 4 cicli compresa call
```

Rispetto a quanto visto prima, si tratta di un semplice adattamento dei valori di pre carica dei contatori e dell' aggiunta finale dell' istruzione **`retlw 0`**.

Subroutine e Macro

Ci si potrebbe chiedere perchè non utilizzare una macro, scrivendo una **`Delay1s`** sotto questa forma. In effetti possiamo sostituire il codice con una macro, come abbiamo visto prima, ma questo non è molto adatto alla situazione.
La motivazione è che ogni volta il sorgente incontra la macro, la sostituisce con la lista delle istruzioni sottintese. Questo porta a due problemi:

- l' intero codice è ripetuto per intero a sostituire ogni macro e quindi si ha una occupazione maggiore di memoria programma
- viene ripetuto così come è stato definito e quindi avremmo ancora il problema delle label duplicate

Il primo punto è normalmente poco sensibile dove la memoria programma del chip è sufficientemente grande; però, nei piccoli Baseline, la sua estensione è limitata e l' uso di grosse macro deve essere valutato con attenzione.

Il secondo punto può essere risolto con una variabile locale nella macro, ma fa parte delle tecniche avanzate che vedremo più avanti. La subroutine è la scelta migliore, anche se pure questa va valutata con attenzione nelle applicazioni su Baseline per i motivi che vedremo subito.

Qui trovate un breve approfondimento sulla questione macro-subroutines.

Il progetto MPLAB

Come per l' esercizio precedente viene fornito l' intero ambiente di lavoro (progetto) per MPLAB per la versione iniziale.

Per le versioni successive, basta sostituire i file sorgenti nel progetto, come già visto, oppure aprire un nuovo progetto, come preferite.

Altre versioni

Possiamo far eseguire la stessa operazione da PIC diversi e notare come si tratti sempre delle stesse azioni. Sono disponibili le versioni per PIC12F508/509, 10F200/202, 16F505/526.

Estensioni

Possiamo proporre alcune variazioni sul tema:

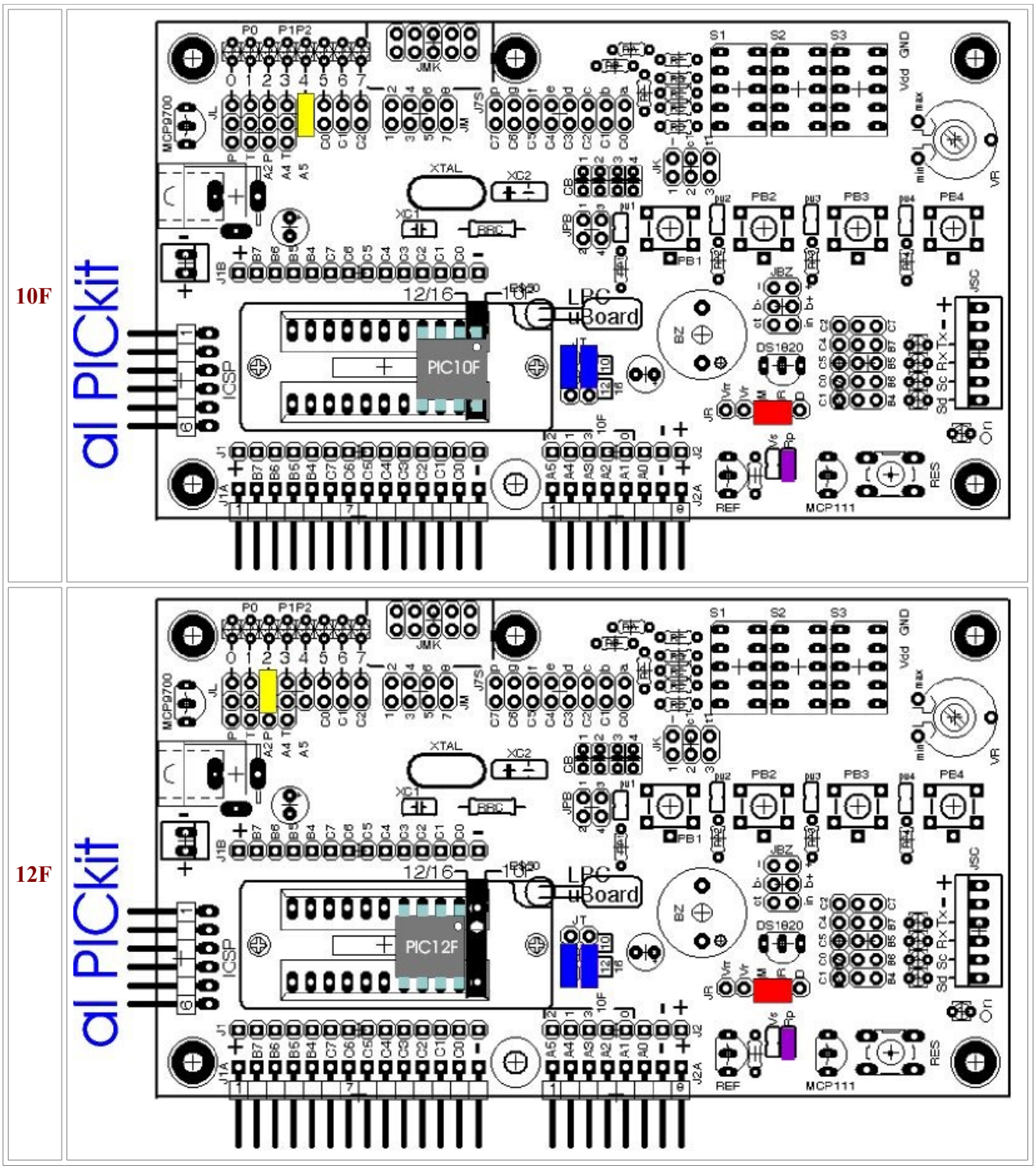
1. variare il tempo di lampeggio del LED
2. far lampeggiare il LED su GP2
3. accendere un LED su GP0 e far lampeggiare un LED su GP2

Se non riuscite a risolvere i "casi", ecco le soluzioni.

1. variando il tempo di attesa della subroutine di ritardo, si varieranno tempi di esecuzione.
   Allegato trovate un esempio per un tempo di 500ms. (***2As_xxx_1.asm***).
   Il piano dei collegamenti sulla **LPCuB** è uguale all' esercizio 1. Per i PIC10F20x, non esistendo un GP5 in questo chip, viene usato GP0.

2. (***2As_xxx_2.asm***) Consultando **la documentazione di 12F519** si rileva che GP2 assume per default la funzione di ingresso del Timer0. Agire come per l' esercitazione precedente. Altrettanto vale per altri Baseline.

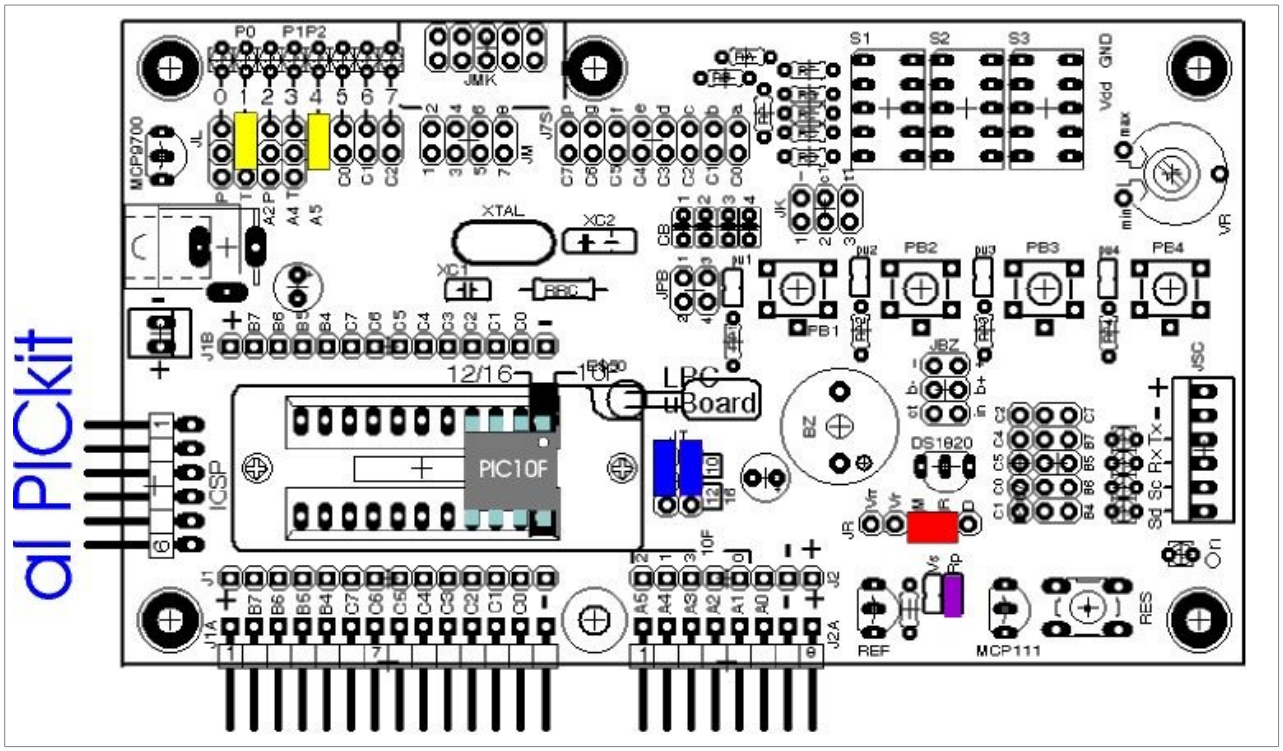
   Qui trovate i jumper da inserire sulla **LPCuB** per:



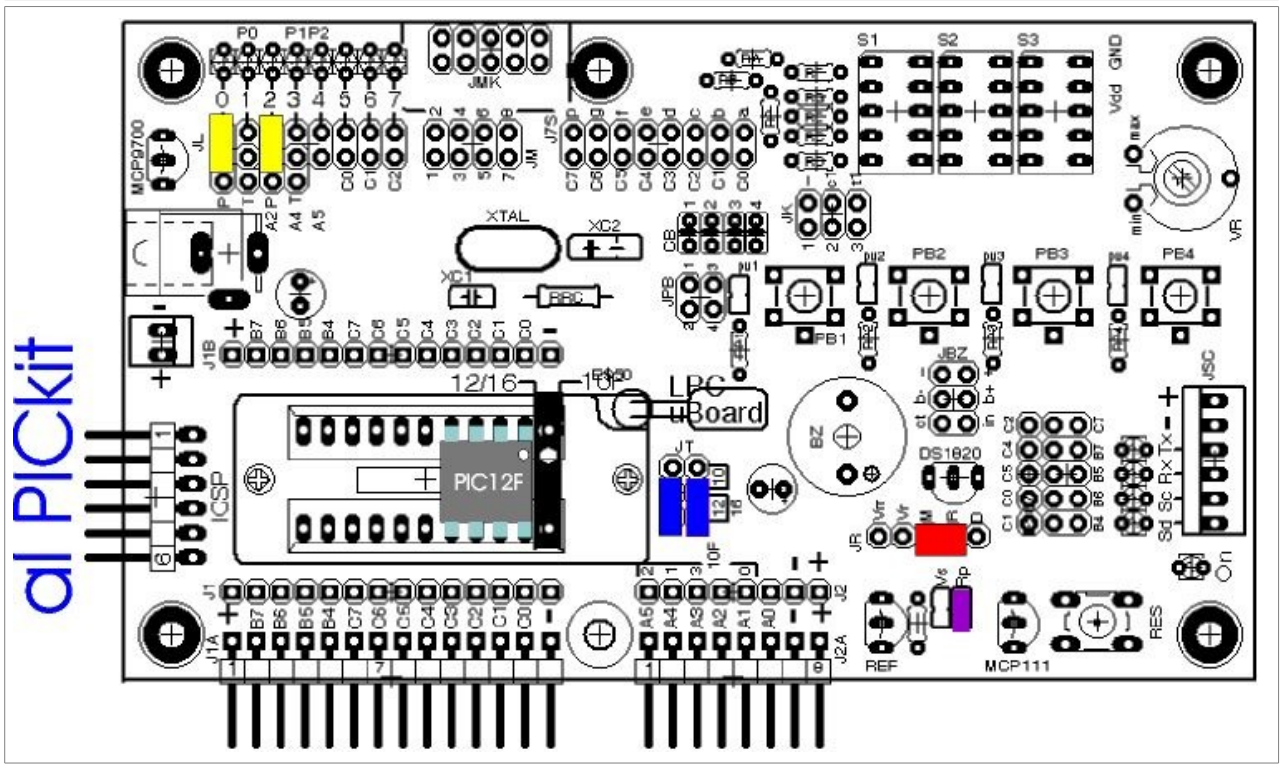
3. Osservare la diversa disposizione per il microcontroller, i jumper **JT** e il jumper di selezione del LED su **JL**

4. (***2As_xxx_2.asm***) Occorre collegare un LED a GP0, analogamente a quanto fatto con GP2.
   Definiti i pin GP0 e GP2 come uscite digitali, basterà portare a livello 1 il bit GP0 e quindi avviare il lampeggio su GP2.
   Qui trovate i jumper da inserire sulla **LPCuB** per:

10F



12F



5. Osservare la diversa disposizione per il microcontroller, i jumper **JT** e il jumper di selezione del LED su **JL**

Nel pacchetto relativo all' esercitazione 2A trovate anche il sorgente di queste versioni.
Come ulteriore esercitazione potete poi provare altre combinazioni di pin e LED.

Conclusioni

Possiamo fare alcune osservazioni a conclusione di queste prime esercitazioni. Da quanto detto e dalla lettura dei sorgenti dovrebbe risultare evidente che:

1. le regole dell' Assembly sono molto semplici, basandosi esclusivamente sugli opcodes e su pochi comandi del compilatore
2. l' impiego di forme simboliche al posto di valori assoluti è un elemento fondamentale
3. la struttura del programma viene determinata dall' impiego di elementi caratteristici derivati dalle funzioni del compilatore (direttive) e dai blocchi logici basilari (macroistruzioni, subroutines). L' uso di questi elementi semplifica il lavoro
4. più una scrittura del sorgente fa uso di simboli, di elementi modulari ed è basata su un template ordinato e più il sorgente è facilmente leggibile e comprensibile

Inoltre, confrontando i sorgenti dei vari PIC esemplificati, possiamo notare che:

1. il sorgente, per componenti della stessa famiglia, è praticamente identico: il programma scritto per un PIC Baseline può essere portato su un altro Baseline con modifiche minimali che dipendono essenzialmente dalle periferiche integrate nel chip.
2. più una scrittura del sorgente fa uso di simboli e di elementi modulari e più il sorgente è facilmente portabile

Vediamo negli esercizi successivi di approfondire questi argomenti.

12F519 - 2Ae_519.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Titolo        :  Corso Assembly & C - Esercitazione 2Ac_519
;                     Lampeggia un LED collegato a GP5 alla
;                     cadenza di 1 secondo.
;                     Genera un errore in compilazione.
;    PIC           : 12F519
;    Supporto      : MPASM
;    Versione      : 1.0
;    Data          : 01-05-2013
;    Ref. hardware :
;    Autore        :  afg
;
;-----------------------------------------------------------------
;
;  Impiego pin :
;  ----------------
;       12F519 @ 8 pin
;
;                    |‾‾\/‾‾|
;              Vdd -|1    8|- Vss
;              GP5 -|2    7|- GP0
;              GP4 -|3    6|- GP1
;         GP3/MCLR -|4    5|- GP2
;                    |______|
;
;    Vdd                 1: ++
;    GP5/OSC1/CLKIN      2: Out  LED alla Vss
;    GP4/OSC2            3:
;    GP3/!MCLR/VPP       4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*****************************************************************
;             DEFINIZIONE DI IMPIEGO DEI PORT
;
;GPIO map
;|  5  |  4  |  3  |  2  |  1  |  0  |
;|-----|-----|-----|-----|-----|-----|
;| LED |     | in  |     |     |     |
;
;#define    GPIO,GP0    ;
;#define    GPIO,GP1    ;
;#define    GPIO,GP2    ;
;#define    GPIO,GP3    ; solo input
;#define    GPIO,P4     ;
#define     LED GPIO,GP5    ; LED tra pin e Vss
;
;#################################################################

        LIST      p=12F519         ; Definizione del processore
```

```
        #include <p12F519.inc>

        radix     dec

;*********************************************************************
;                          CONFIGURAZIONE
;
; Oscillatore interno, no WDT, no CP, pin4=MCLR
 __config  _IntRC_OSC & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF &
_MCLRE_ON


;*********************************************************************
;                        MEMORIA RAM
; general purpose RAM
        CBLOCK 0x07         ; inizio area RAM
      d1,d2,d3              ; contatori per ritardo
        ENDC

;*********************************************************************
;                       LOCAL MACROS
;
;Comandi per il LED
LED_ON    MACRO
        bsf     LED
          ENDM
LED_OFF   MACRO
        bcf     LED
          ENDM

 PAGE

;*********************************************************************
;                         RESET ENTRY
;
; Reset Vector
        ORG     0x00

 ; calibrazione oscillatore interno
        movwf   OSCCAL

;*********************************************************************
;                        MAIN PROGRAM
;
MAIN:
; inizializzazioni al reset
        clrf    GPIO            ; preset GPIO latch a 0

; GP5 output
; TRISGPIO        --011111        GP5 out
        movlw   b'11011111'
        tris    GPIO            ; al registro direzione

mainloop:               ;<<-----<<<--------------<<----|
; accende LED           ;                              |
        LED_ON          ;                              |
                        ;                              |
```

```
; attesa 1 s                                                    |
Delay1s:                ; 999997 cycles                        |
        movlw   0x08    ; inizializza counter                  |
        movwf   d1      ;                                      |
        movlw   0x2F    ;                                      |
        movwf   d2      ;                                      |
        movlw   0x03    ;                                      |
        movwf   d3      ;                                      |
Delay1s_0               ; loop di conteggio                    |
        decfsz  d1, f   ; d1=d1-1                              |
        goto    $+2     ;                                      |
        decfsz  d2, f   ; d2=d2-2                              |
        goto    $+2     ;                                      |
        decfsz  d3, f   ; d3=d3-1                              |
        goto    Delay1s_0 ;                                    |
; fine 999997 cicli - ora somma i 3 mancanti                   |
        goto    $+1     ;3 cycles                              |
        nop             ;                                      |
                        ;                                      |
; spegne LED            ;                                    loop
        LED_OFF         ;                                      |
                        ;                                      |
; attesa 1 s                                                   |
Delay1s:                ; 999997 cycles                        |
        movlw   0x08    ; inizializza counter                  |
        movwf   d1      ;                                      |
        movlw   0x2F    ;                                      |
        movwf   d2      ;                                      |
        movlw   0x03    ;                                      |
        movwf   d3      ;                                      |
Delay1s_0               ; loop di conteggio                    |
        decfsz  d1, f   ; d1=d1-1                              |
        goto    $+2     ;                                      |
        decfsz  d2, f   ; d2=d2-2                              |
        goto    $+2     ;                                      |
        decfsz  d3, f   ; d3=d3-1                              |
        goto    Delay1sa_0 ;                                   |
; fine 999997 cicli - ora somma i 3 mancanti                   |
        goto    $+1     ;3 cicli                               |
        nop             ;                                      |
                        ;                                      |
; loop                                                         |
        goto    mainloop ;>>------>>>-------------->>----|

;*****************************************************************
;                           THE END
;*****************************************************************
        END
```

12F519 - 2As_519.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Titolo        :  Corso Assembly & C - Esercitazione 2As_519
;                     Lampeggia un LED collegato a GP5 alla
;                     cadenza di 1 secondo.
;                     Subroutine.
;    PIC           : 12F519
;    Supporto      : MPASM
;    Versione      : 1.0
;    Data          : 01-05-2013
;    Ref. hardware :
;    Autore        :  afg
;
;-----------------------------------------------------------------
;
;  Impiego pin :
;  ----------------
;       12F519 @ 8 pin
;
;                    |__\/__|
;              Vdd -|1    8|- Vss
;              GP5 -|2    7|- GP0
;              GP4 -|3    6|- GP1
;         GP3/MCLR -|4    5|- GP2
;                    |______|
;
;    Vdd                 1: ++
;    GP5/OSC1/CLKIN      2: Out  LED alla Vss
;    GP4/OSC2            3:
;    GP3/!MCLR/VPP       4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*****************************************************************
;             DEFINIZIONE DI IMPIEGO DEI PORT
;
;GPIO map
;|  5  |  4  |  3  |  2  |  1  |  0  |
;|-----|-----|-----|-----|-----|-----|
;| LED |     | in  |     |     |     |
;
;#define    GPIO,GP0    ;
;#define    GPIO,GP1    ;
;#define    GPIO,GP2    ;
;#define    GPIO,GP3    ; solo input
;#define    GPIO,P4     ;
#define     LED GPIO,GP5    ; LED tra pin e Vss
;
;#################################################################

        LIST      p=12F519         ; Definizione del processore
```

```
        #include <p12F519.inc>

        radix     dec

;*********************************************************************
;                          CONFIGURAZIONE
;
; Oscillatore interno, no WDT, no CP, pin4=MCLR;
 __config  _IntRC_OSC & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF &
_MCLRE_ON


;*********************************************************************
;                         MEMORIA RAM
;
; general purpose RAM
        CBLOCK 0x07         ; inizio area RAM
      d1,d2,d3              ; contatori per ritardo
        ENDC

;*********************************************************************
;                       LOCAL MACROS
;
; Comandi per il LED
LED_ON    MACRO
        bsf     LED
          ENDM
LED_OFF   MACRO
        bcf     LED
          ENDM


 PAGE

;*********************************************************************
;                          RESET ENTRY
;
; Reset Vector
        ORG     0x00

 ; calibrazione oscillatore interno
        movwf   OSCCAL


;*********************************************************************
;                         MAIN PROGRAM
;
MAIN:
; inizializzazioni al reset
     clrf    GPIO         ; preset GPIO latch a 0

; TRISGPIO     --011111 GP5 out
     movlw   b'11011111'
     tris    GPIO         ; al registro direzione

     goto    mainloop
```

```
; tabella salti alle subroutines
Delay1s  goto    Dly1s

;=====================================================================
mainloop:               ;<<--<<<--|
; accende LED           ;         |
     LED_ON             ;         |
                        ;         |
; attesa 1s                       |
     call    Delay1s    ;         |
                        ;         |
; spegne LED                   loop
     LED_OFF            ;         |
                        ;         |
; attesa 1s                       |
     call    Delay1s    ;         |
                        ;         |
; loop                  ;         |
     goto    mainloop   ;>>-->>---|

;=====================================================================
;=                      SUBROUTINES                                  =
;=====================================================================
; Delay = 1 secondo @ Clock = 4 MHz
; Fosc/4 = 1 us , 1 secondo = 1000000 cicli
Dly1s:                   ; 999990 cicli
        movlw   0x07     ; inizializza counter
        movwf   d1       ;
        movlw   0x2F     ;
        movwf   d2       ;
        movlw   0x03     ;
        movwf   d3       ;
Dly1s_0:                 ; loop di conteggio
        decfsz  d1, f    ; d1=d1-1
        goto    $+2      ;
        decfsz  d2, f    ; d2=d2-2
        goto    $+2      ;
        decfsz  d3, f    ; d3=d3-1
        goto    Dly1s_0  ;
; fine 999990 cicli - ora somma i 10 mancanti
        goto    $+1      ; 6 cicli
        goto    $+1      ;
        goto    $+1      ;
        retlw   0        ; 4 cicli compresa call

;*********************************************************************
;                            THE END
        END
```

12F508/509 - 2As_5089.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Titolo        :  Corso Assembly & C - Esercitazione 2As_5089
;                     Lampeggia un LED collegato a GP5 alla
;                     cadenza di 1 secondo.
;                     Subroutine.
;    PIC           : 12F508/9
;    Supporto      : MPASM
;    Versione      : V.519-1.0
;    Data          : 01-05-2013
;    Ref. hardware :
;    Autore        :  afg
;
;-----------------------------------------------------------------
;
;  Impiego pin :
;  ----------------
;       12F508/9 @ 8 pin
;
;                   |__\/__|
;              Vdd -|1    8|- Vss
;              GP5 -|2    7|- GP0
;              GP4 -|3    6|- GP1
;         GP3/MCLR -|4    5|- GP2
;                   |______|
;
;    Vdd                 1: ++
;    GP5/OSC1/CLKIN      2: Out  LED alla Vss
;    GP4/OSC2            3:
;    GP3/!MCLR/VPP       4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*****************************************************************
;             DEFINIZIONE DI IMPIEGO DEI PORT
;
;GPIO map
;|  5  |  4  |  3  |  2  |  1  |  0  |
;|-----|-----|-----|-----|-----|-----|
;| LED |     | in  |     |     |     |
;
;#define    GPIO,GP0    ;
;#define    GPIO,GP1    ;
;#define    GPIO,GP2    ;
;#define    GPIO,GP3    ; solo input
;#define    GPIO,P4     ;
#define     LED GPIO,GP5    ; LED tra pin e Vss
;
;#################################################################
; scelta del processore
 #ifdef  __12F509
```

```
        LIST      p=12F509         ; Definizione del processore
        #include <p12F509.inc>
 #endif
 #ifdef  __12F508
       LIST      p=12F508          ; Definizione del processore
        #include <p12F508.inc>
 #endif

        radix     dec

;*****************************************************************
;                         CONFIGURAZIONE
;
 Oscillatore interno, no WDT, no CP, pin4=GP3
 __config  _IntRC_OSC & _WDT_OFF & _CP_OFF & _MCLRE_ON


;*****************************************************************
;                         MEMORIA RAM
;
; general purpose RAM
        CBLOCK 0x07          ; inizio area RAM
      d1,d2,d3               ; contatori per ritardo
        ENDC


;*****************************************************************
;                         LOCAL MACROS
; Comandi per il LED
LED_ON    MACRO
        bsf     LED
          ENDM
LED_OFF   MACRO
        bcf     LED
          ENDM

 PAGE
;*****************************************************************
;                          RESET ENTRY
;
; Reset Vector
        ORG     0x00

 ; calibrazione oscillatore interno
        movwf   OSCCAL

;*****************************************************************
;                         MAIN PROGRAM
;
MAIN:
; inizializzazioni al reset
     clrf    GPIO         ; preset GPIO latch a 0

; TRISGPIO     --011111   GP5 out
     movlw   b'11011111' ; maschera direzione PORT
     tris    GPIO         ; al registro direzione
```

```
     goto    mainloop

; tabella salti alle subroutines
Delay1s  goto    Dly1s

;=====================================================================
mainloop:             ;<<--<<<--|
; accende LED         ;         |
  LED_ON              ;         |
                      ;         |
; attesa 1s                      |
  call    Delay1s     ;         |
                      ;         |
; spegne LED                   loop
  LED_OFF             ;         |
                      ;         |
; attesa 1s                      |
  call    Delay1s     ;         |
                      ;         |
; loop                ;         |
  goto    mainloop    ;>>-->>---|

;=====================================================================
;=                       SUBROUTINES                                 =
;=====================================================================
; Delay = 1 secondo @ Clock = 4 MHz
; Fosc/4 = 1 us , 1 secondo = 1000000 cicli
Dly1s:                    ; 999990 cicli
        movlw   0x07      ; inizializza counter
        movwf   d1        ;
        movlw   0x2F      ;
        movwf   d2        ;
        movlw   0x03      ;
        movwf   d3        ;
Dly1s_0:                  ; loop di conteggio
        decfsz  d1, f     ; d1=d1-1
        goto    $+2       ;
        decfsz  d2, f     ; d2=d2-2
        goto    $+2       ;
        decfsz  d3, f     ; d3=d3-1
        goto    Dly1s_0   ;
; fine 999990 cicli - ora somma i 10 mancanti
        goto    $+1       ; 6 cicli
        goto    $+1       ;
        goto    $+1       ;
        retlw   0         ; 4 cicli compresa call

;*********************************************************************
;                          THE END
        END
```

10F200/202 - 2As_20x.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Titolo        :  Corso Assembly & C - Esercitazione 1As_20x
;                     Lampeggia un LED collegato a GP0 alla
;                     cadenza di 1 secondo.

;    PIC           :  10F200/2
;    Supporto      :  MPASM
;    Versione      :  1.0
;    Data          :  01-05-2013
;    Ref. hardware :
;    Autore        :  afg
;
;
;  Impiego pin :
;  ----------------
;     10F200/202 @ 8 pin DIP       10F200/202 @ 6 pin SOT-23
;
;               |__\/__|                 *______|
;           NC -|1    8|- GP3        GP0 -|1    6|- GP3
;          Vdd -|2    7|- Vss        Vss -|2    5|- Vdd
;          GP2 -|3    6|- NC         GP1 -|3    4|- GP2
;          GP1 -|4    5|- GP0              |______|
;               |______|
;
;                       DIP  SOT
;    NC                  1:     nc
;    Vdd                 2:  5: ++
;    GP2/T0CKI/FOSC4     3:  4:
;    GP1/ICSPCLK         4:  3:
;    GP0/ICSPDAT         5:  1: Out LED alla Vss
;    NC                  6:     nc
;    Vss                 7:  2: --
;    GP3/MCLR/VPP        8:  6:
;
;*****************************************************************
;            DEFINIZIONE DI IMPIEGO DEI PORT
;GPIO map
; |  3  |  2  |  1  |  0  |
; |-----|-----|-----|-----|
; | in  |     |     | LED |
;
#define    LED GPIO,GP0   ; LED tra pin e Vss
;#define    GPIO,GP1   ;
;#define    GPIO,GP2   ;
;#define    GPIO,GP3   ; solo input
;
;#################################################################
 #ifdef __10F200
       LIST      p=10F200     ; definizione del processore
       #include <p10F200.inc>
 #endif
 #ifdef __10F202
```

```
        LIST      p=10F202     ; definizione del processore
        #include <p10F202.inc>
 #endif

        radix     dec          ; default numeri decimali

;*****************************************************************
;                           CONFIGURAZIONE
;
; No WDT, no CP, pin4=GP3;
  __config  _CP_OFF & _MCLRE_OFF  & _WDT_OFF

;*****************************************************************
;                           LOCAL MACROS
;
; Comandi per il LED
LED_ON    MACRO
        bsf     LED
          ENDM
LED_OFF   MACRO
        bcf     LED
          ENDM

 PAGE
;*****************************************************************
;                            RESET ENTRY
;
; Reset Vector
RESET_VECTOR   ORG     0x00

 ; calibrazione oscillatore interno
        movwf   OSCCAL

;*****************************************************************
;                         MAIN PROGRAM
;
MAIN:
; inizializzazioni al reset
        clrf    GPIO            ; preset GPIO latch a 0

; TRISGPIO        --111110     GP0 out
        movlw   b'11111110'
        tris    GPIO            ; al registro direzione

     goto    mainloop

; tabella salti alle subroutines
Delay1s  goto    Dly1s

;=================================================================
mainloop:            ;<<--<<<--|
; accende LED        ;         |
  LED_ON             ;         |
                     ;         |
; attesa 1s                     |
  call    Delay1s    ;         |
                     ;         |
```

```
; spegne LED                    loop
  LED_OFF             ;          |
                      ;          |
; attesa 1s                      |
  call    Delay1s     ;          |
                      ;          |
; loop                ;          |
  goto    mainloop    ;>>-->>---|

;=====================================================================
;=                        SUBROUTINES                                =
;=====================================================================
; Delay = 1 secondo @ Clock = 4 MHz
; Fosc/4 = 1 us , 1 secondo = 1000000 cicli
Dly1s:                      ; 999990 cicli
        movlw   0x07        ; inizializza counter
        movwf   d1          ;
        movlw   0x2F        ;
        movwf   d2          ;
        movlw   0x03        ;
        movwf   d3          ;
Dly1s_0:                    ; loop di conteggio
        decfsz  d1, f       ; d1=d1-1
        goto    $+2         ;
        decfsz  d2, f       ; d2=d2-2
        goto    $+2         ;
        decfsz  d3, f       ; d3=d3-1
        goto    Dly1s_0     ;
; fine 999990 cicli - ora somma i 10 mancanti
        goto    $+1         ; 6 cicli
        goto    $+1         ;
        goto    $+1         ;
        retlw   0           ; 4 cicli compresa call

;*********************************************************************
;                             THE END
        END
```

16F526/505 - 2As_526.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Titolo        :  Corso Assembly & C - Esercitazione 2As_20x
;                     Lampeggia un LED collegato a GP0 alla
;                     cadenza di 1 secondo.
;                     Subroutine.
;    PIC           : 16F526-16F505
;    Supporto      : MPASM
;    Versione      : 1.0
;    Data          : 01-05-2013
;    Ref. hardware :
;    Autore        :  afg
;
;-----------------------------------------------------------------
;
;  Impiego pin :
;  ---------------
;       16F505 - 16F526 @ 14 pin
;
;                   |__\/__|
;              Vdd -|1   14|- Vss
;              RB5 -|2   13|- RB0
;              RB4 -|3   12|- RB1
;       RB3/MCLR -|4   11|- RB22
;              RC5 -|5   10|- RC0
;              RC4 -|6    9|- RC1
;              RC3 -|7    8|- RC2
;                   |______|
;
;    Vdd                       1: ++
;    RB5/OSC1/CLKIN            2: Out  LED alla Vss
;    RB4/OSC2/CLKOUT           3:
;    RB3/!MCLR/VPP             4:
;    RC5/T0CKI                 5:
;    RC4/[C2OUT]               6:
;    RC3                       7:
;    RC2/[Cvref]               8:
;    RC1/[C2IN-]               9:
;    RC0/[C2IN+]              10:
;    RB2/[C1OUT/AN2]          11:
;    RB1/[C1IN-/AN1/]ICSPC    12:
;    RB0/[C1IN+/AN0/]ICSPD    13:
;    Vss                      14: --
;
;    [ ] solo 16F526
;
;*****************************************************************
;===============================================================
;            DEFINIZIONE DI IMPIEGO DEI PORT
;
;PORTC non utilizzato
;
```

```
; PRTB map
; |  5  |  4  |  3  |  2  |  1  |  0  |
; |-----|-----|-----|-----|-----|-----|
; | LED |     |     |     |     |     |
;
#define   LED  PORTB,RB5    ; LED tra pin e Vss
;#define       PORTB,RB3
;#define       PORTB,RB2
;#define       PORTB,RB1
;#define       PORTB,RB0

;*****************************************************************
;                          SELEZIONE PROCESSORE
 #ifdef __16F526
      LIST p=16F526
      #include <p16F526.inc>
  #endif
  #ifdef __16F505
      LIST p=16F505
      #include <p16F505.inc>
 #endif

;*****************************************************************
;                           CONFIGURAZIONE
;
 #ifdef __16F526
; Oscillatore interno, 4MHz, no WDT, no CP, no MCLR
 __config _IntRC_OSC & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF &
MCLRE_OFF
 #endif

 #ifdef __16F505
; Oscillatore interno, 4MHz, no WDT, no CP, no MCLR
 __config _IntRC_OSC_RB4 & _WDT_OFF & _CP_OFF & _MCLRE_OFF
 #endif

;*****************************************************************
;=================================================================
;*****************************************************************
;                         MEMORIA RAM
;
; general purpose RAM
        CBLOCK 0x10         ; inizio area RAM
      d1,d2,d3              ; contatori per ritardo
      ENDC

;*****************************************************************
;                       LOCAL MACROS
;
; Comandi per il LED
LED_ON    MACRO
        bsf     LED
          ENDM
LED_OFF   MACRO
        bcf     LED
          ENDM
```

```
;*********************************************************************
;                          RESET ENTRY
;
; Reset Vector
        ORG     0x00

 ; calibrazione oscillatore interno
        movwf   OSCCAL

;*********************************************************************
;                         MAIN PROGRAM
;
MAIN:
; inizializzazioni al reset
     clrf    PORTB       ; preset port latch a 0

; TRISB        --011111  RB5 out
     movlw   b'11011111'
     tris    PORTB       ; al registro direzione

     goto    mainloop

; tabella salti alle subroutines
Delay1s  goto    Dly1s

;=====================================================================
mainloop:               ;<<--<<<--|
; accende LED           ;         |
     LED_ON             ;         |
                        ;         |
; attesa 1s                       |
     call    Delay1s    ;         |
                        ;         |
; spegne LED                   loop
     LED_OFF            ;         |
                        ;         |
; attesa 1s                       |
     call    Delay1s    ;         |
                        ;         |
; loop                  ;         |
     goto    mainloop   ;>>-->>---|

;=====================================================================
;                       SUBROUTINES
;
; Delay = 1 secondo @ Clock = 4 MHz
; Fosc/4 = 1 us , 1 secondo = 1000000 cicli
Dly1s:                  ; 999990 cicli
        movlw   0x07    ; inizializza counter
        movwf   d1      ;
        movlw   0x2F    ;
        movwf   d2      ;
        movlw   0x03    ;
        movwf   d3      ;
Dly1s_0:                ; loop di conteggio
```

```
        decfsz  d1, f     ; d1=d1-1
         goto   $+2       ;
        decfsz  d2, f     ; d2=d2-2
         goto   $+2       ;
        decfsz  d3, f     ; d3=d3-1
         goto   Dly1s_0   ;
; fine 999990 cicli - ora somma i 10 mancanti
        goto    $+1       ; 6 cicli
        goto    $+1       ;
        goto    $+1       ;
        retlw   0         ; 4 cicli compresa call

;*****************************************************************
;                              THE END
        END
```