Esercitazioni C Enhanced Midrange

8 – Timer0 e Interrupt

Fino ad ora abbiamo visto essenzialmente come utilizzare i semplici I/O digitali e le funzioni associate, ma abbiamo introdotto alcuni moduli come l'oscillatore interno INTOSC.
Iniziamo ora a vedere i particolari di altri moduli, a partire dai timer.

Un timer è un sistema di conteggio, alimentato da un clock selezionabile e prescalabile.
I timer dei PIC sono “a crescere”, ovvero il conteggio parte da 0 e conta fino al massimo della capacità del registro contatore: per timer a 8 bit è, appunto, un byte, con la capacità massima di 255; per i timer a 16bit il registro di conteggio occupa due bytes, per un massimo di 65535.

Una particolarità dei timer è quella di determinare all'overflow del contatore un evento di richiesta di interruzione (interrupt). Questo permette di stabilire una precisa cadenza di base indispensabile per numerose attività, oppure per determinare temporizzazioni e ritardi programmati.

In generale, si parla di timer/counter in quanto i moduli di conteggio possono essere alimentati sia con un clock interno, che ha un periodo determinato, per realizzare funzioni di temporizzazione, sia con un segnale esterno per realizzare funzioni di conteggio.

Negli Enhanced sono disponibili diversi timer a 8 e a 16 bit: vediamo per primo il **Timer0** che è presente in tutti i PIC.

8.1 – Timer0

Timer0 é presente in tutti i PIC a 8 bit, con struttura analoga e compatibile per quanto riguarda il software (solamente nei PIC18F ha la possibilità di estendere il conteggio a 16bit).

Le caratteristiche di questo timer sono le seguenti:

- **contatore a 8 bit**: come solito per Microchip, si tratta di un contatore "a crescere", cioè che conta in salita da 00 a 0xFF (255 decimale), incrementando di 1 ad ogni impulso del clock e che va in overflow al 256esimo impulso, ripartendo da 0.
- **clock interno o esterno**: può essere derivato dal clock principale (**Fosc/4**) o da un segnale esterno applicato al pin **T0CKI** (**T**imer**0** **C**loc**K** **I**nput).
- **fronte del clock programmabile**: è possibile programmare il fronte di commutazione del livello del segnale di ingresso estreno per cui il contatore avanza.
- **prescaler 2/4/8/16/32/64/128/256**: il prescaler divide per il fattore indicato il segnale di ingresso, per poter ottenere una ampia gamma di tempi.
- **sorgente di interrupt non periferico**: il TIMER0 è fonte di interrupt non periferico, abilitabile da programma.

Timer 0 non può essere spento se non sospendendo il clock.

Da notare che nei Baseline e Midrange il prescaler è in comune con il WDT e può essere assegnato solamente ad una delle due periferiche; questa situazione riduce la flessibilità di impiego in alcune applicazioni. Invece, **negli Enhanced, il Timer0 ha un prescaler proprio** e non richiede le manovre di commutazione dei PIC precedenti.

Il timer è governato da alcuni bit nel registro **`OPTION_REG`**, di cui abbiamo già parlato nelle esercitazioni precedenti a riguardo dei weak pull-up.

L'aspetto di **`OPTION_REG`** è questo (sono evidenziati i bit che interessano Timer0):

bit	7	6	5	4	3	2	1	0
`OPTION`	**`WPUEN`**	**`INTEDG`**	**`TMR0CS`**	**`TMR0SE`**	**`PSA`**	**`PS2`**	**`PS1`**	**`PS0`**

- **`TMR0CS`** consente di selezionare il clock del timer
  - 0 = clock interno **`Fosc/4`**
  - 1 = clock esterno sul pin **`T0CKI`**
- **`TMR0SE`** consente di selezionare il fronte di commutazione del clock **esterno** su cui far avanzare il conteggio
  - 0 = transizione alto-basso (falling edge)
  - 1 = transizione basso- alto (rising edge)
- **`PSA`** attiva o disattiva il prescaler del timer
  - 0 = prescaler inserito
  - 1 = prescaler disattivato

- **PS<2:0>** sono i bit che consentono di selezionare il fattore di divisione del prescaler

```
000   1:2
001   1:4
010   1:8
011   1:16
100   1:32
101   1:64
110   1:128
111   1:256
```

Va notato che al POR il registro OPTION ha tutti i bit settati.
Quindi, la configurazione di default del Timer0 è:

- **clock** dal pin esterno **T0CKI**
- conteggio sul fronte di discesa
- **prescaler non attivato**
- prescaler 1:256

Se l'applicazione richiede una configurazione diversa, occorre modificare da programma i vari bit.

Ad esempio, per disporre del timer con clock interno e prescaler 1:32

```
OPTION_REGbits.TMR0CS = 0;   // clock interno
OPTION_REGbits.PSA = 0;      // prescaler abilitato
OPTION_REGbits.PS = 0b100;   // 1:32
```

Il Timer0 non è arrestabile: fino che il clock è presente, il conteggio continua.
Altri timer dispongo di interruttore di abilitazione, ma il Timer0 si arresta solo sospendendo il clock.

Il registro di conteggio del timer è **TMR0** a 8 bit, che è leggibile e scrivibile. Invece, **NON è accessibile il contenuto del prescaler.**

Scrivendo in **TMR0**, l'incremento del contatore è bloccato per due cicli di assestamento e di questo va tenuto conto nelle applicazioni dove è richiesta una temporizzazione precisa.
A questo proposito, dovrebbe essere ovvio che la **precisione e stabilità del timer è la stessa del clock.**

Il tempo ottenibile dal **Timer0** dipende, ovviamente, dalla frequenza del clock e dal valore di partenza del contatore del timer. Una formula generale è questa:

$$Period = (256 - TMR0)*(1/clock)*(Prescaler)$$

A esempio, pre caricando il contatore con 100 ed usando il clock **FOSC/4=1MHz**, con prescaler 1:64 si ottiene un periodo di

$$Period = (256 - 100)*(1)*(64) = 9984us$$

In effetti, se utilizziamo un valore per pre caricare il registro **TMR0** si deve considerare che il conteggio si arresta per due cicli, quindi, per una temporizzazione esatta, si dovrà sottrarre 2 al valore pre caricato, quindi 98 e non 100.

Va notato che il contatore TMR0 del Timer0 non si ricarica automaticamente all'overflow: una volta raggiunto il valore massimo 0xFF, il contatore si azzera e riparte da questo valore.

Dove richiesto, va ricaricato un offset dal programma per mantenere la stessa cadenza prefissata.

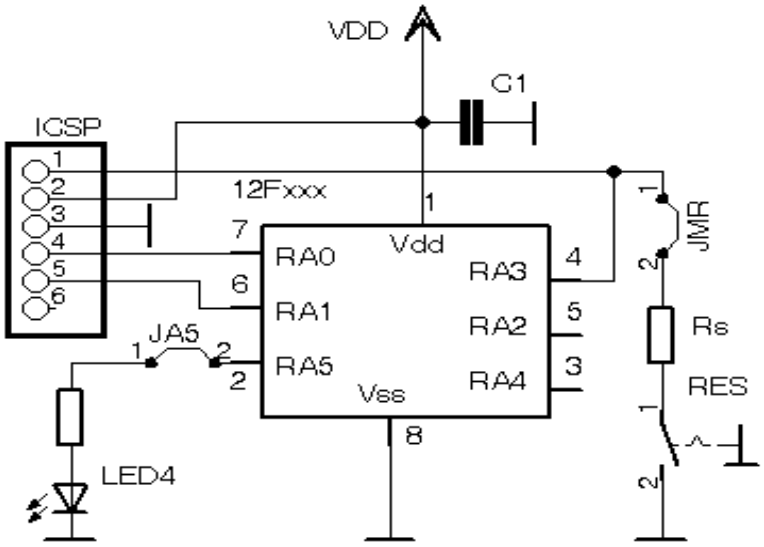
Altre informazioni sui Timer.

Vediamo una possibile applicazione esemplificativa.

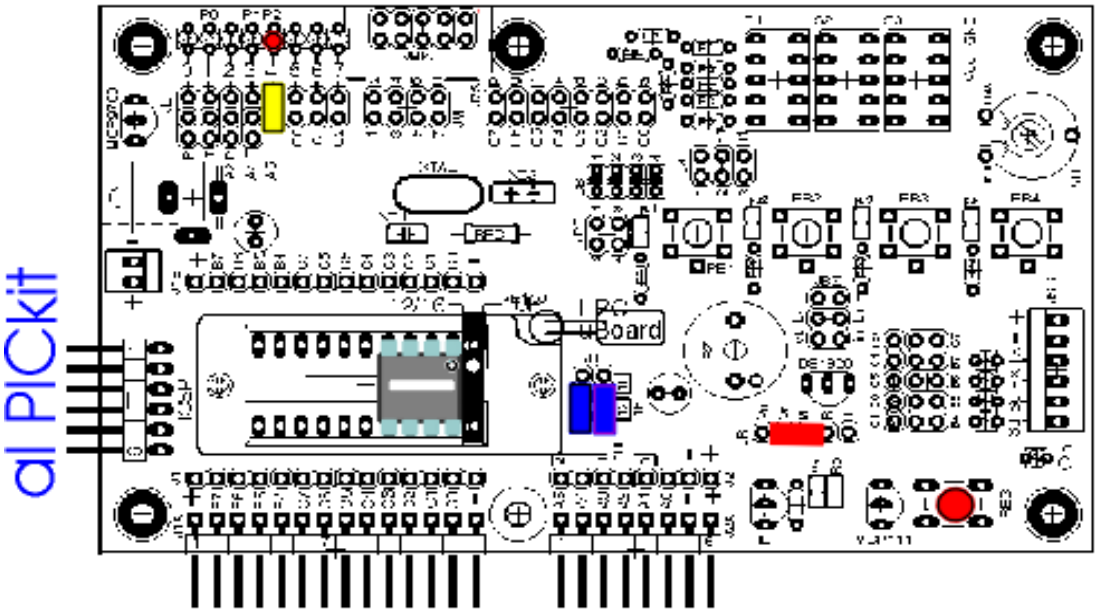
8.2 – Blinking LED con TMR0

Abbiamo visto come far lampeggiare un LED con una cadenza determinata. Nell'esercitazione 2 abbiamo utilizzato la macro **`__delay_ms()`** fornita dal compilatore XC8.
Vediamo ora come eseguire la stessa funzione utilizzando come elemento di tempo il Timer0.

Utilizziamo la configurazione minimale della prima esercitazione (le indicazioni dei componenti ricalcano quelle della scheda **LPCuB** (www)).



Ricordiamo che **`MCLR`** ha il suo pull-up integrato e non ne serve uno esterno.
Sulla **LPCuB** (www)



Non serve altro.

Per quanto riguarda il programma, il tempo di overflow del timer è facilmente calcolabile.
La frequenza degli eventi di overflow è:

Fout = XTAL_CLK / 4 * (256-TMR0) * prescaler

per cui il periodo di evento di overflow è:

Tout = 1 / Fout

Con un clock di 500kHz, il ciclo di istruzione è:

500kHz/4 = 125kHz, cioè un periodo di **8us**.

Ogni 8us il contatore TMR0 avanza di una unità. Essendo a 8 bit, occorrono 256 impulsi per arrivare all'overflow. Quindi il periodo di overflow sarebbe:

8us * 256 = 2048us

Se utilizziamo il **prescaler 1:256**, **il contatore avanzerà di un passo ogni 256 impulsi di clock**, quindi il periodo sarà:

2048 * 256 = 524288us = 524.288ms

Se vogliamo avvicinarci al valore di 500ms, dobbiamo **ridurre** il numero contenuto nel contatore TMR0. Se **carichiamo TMR0 con il valore 12**, il conteggio partirà da questo valore per cui l'overflow sarà ottenuto dopo:

256-12 = 244 impulsi

Quindi il periodo di overflow diventa :

(***8us * 244) * 256 = 499712us = 499.712ms***

Questo è il valore più vicino a 500ms che posiamo ottenere con le condizioni applicate e che per la nostra applicazione è più che adeguato.

Dato che **vanno persi 2 cicli** ad ogni ricarica di TMR0, il valore reale con cui caricheremo il contatore sarà ***12 – 2 = 10***.

Tutti questi conti, una volta capito il meccanismo con cui funziona il timer, NON è necessario ripeterli, dato che possiamo sfruttare con vantaggio uno dei tanti ***Timer0 Calculator*** offerti dal Web, dal quale otteniamo i parametri necessari per avere dal timer un periodo voluto.
Il calcolatore ci fornisce anche un suggerimento sulle istruzioni da fornire al sorgente

```
// Timer0 Registers: Prescaler=1:256; TMR0 Preset=12; Period=499,712 ms
OPTION_REGbits.TMR0CS = 0;   // Clock Source = Internal Clock (CLKO)
OPTION_REGbits.PSA  = 0;     // bit 3 Prescaler to Timer0
OPTION_REGbits.PS  = 0b111;  // PS2:PS0: Prescaler 1:256
TMR0 = 12;                   // preset for counter
```

Osserviamo che con un clock di frequenza maggiore, questo periodo non sarebbe ottenibile: già con 1MHz di clock il tempo massimo è di circa 262ms.

Il timer, che si era avviato subito **dopo il POR** (e di conseguenza **TMR0** contiene un valore indeterminato), viene pre caricato con 12 (decimale) e ricomincia a contare da questo valore fino ad arrivare all'overflow (**0xFF** impulsi di clock -12 +1).

Possiamo verificare l'overflow testando il bit **TMR0IF** del registro **INTCON**, che abbiamo già visto parlando di IOC.

Dato che **nessuno switch di abilitazione dell'interrupt è abilitato**, l'overflow avrà come risultato solamente il settaggio del bit **IF**.

In coincidenza dell'evento, invertiamo lo stato del LED e cancelliamo il flag **IF** per poter rilevare il prossimo overflow.
Sarà, inoltre, necessario ricaricare il contatore con il valore voluto per avere una cadenza costante. Siccome il timer si arresta per due cicli quando viene scritto il registro di conteggio **TMR0**, occorrerà adeguare il valore di pre carica a 12-2=10.

Ripetiamo il ciclo indefinitamente con il solito `while(1).`

es8C

```
/*****************************************************************
*----------------------------------------------------------------
*    Titolo        :  C Enhanced - es8C
*                     Un LED collegato tra R5 e Vss lampeggia una
*                     volta al secondo con Timer0.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  12F1571/2
*    Autore        :  afg
*
*----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       12F1571/2 @ 8 pin
*
*                   |__\/__|
*              Vdd -|1    8|- Vss
*              RA5 -|2    7|- RA0
*              RA4 -|3    6|- RA1
*         RA3/MCLR -|4    5|- RA2
*                   |______|
*
*    Vdd               1: ++
*    RA5/OSC1/CLKIN    2: Out LED
*    RA4/OSC2/CLKOUT   3: In
*    RA3/!MCLR/VPP     4: MCLR
*    RA2               5: In
*    RA1/ICSPCLK       6: In
*    RA0/ICSPDAT       7: In
*    Vss               8: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/

// intosc, no clockout, no wdt, no pwrt, mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1572_0.h"

#include <xc.h>
#define Led4 LATAbits.LATA5

/************************ MAIN PROGRAM ****************************/
void main(){

    // Inizializza I/O
    ANSELA=0;               // no analogiche
    LATAbits.LATA5  = 1;    // preset LED on high
    TRISAbits.TRISA5 = 0;   // RA5 come output

  // abilitiamo wpu per protezione pin non usati
   OPTION_REGbits,nWPUEN = 0;

  // Timer0 Registers: Prescaler=1:256; TMR0 Preset=12; Period=499,712 ms
   OPTION_REGbits.TMR0CS = 0;   // Clock Source = Internal Clock (CLKO)
```

```
   OPTION_REGbits.PSA  = 0;     // bit 3 Prescaler to Timer0
   OPTION_REGbits.PS  = 0b111;  // PS2:PS0: Prescaler 1:256
   TMR0 = 10;                   // preset for counter
   TMR0IF=0 ;                   // clear flag

// Loop principale
   while (1) {
     while (TMR0IF==0);         // attesa overflow
     TMR0 = 10;
     TMR0IF = 0 ;
     Led4 = ! Led4;             // toggle LED
   }
}
```

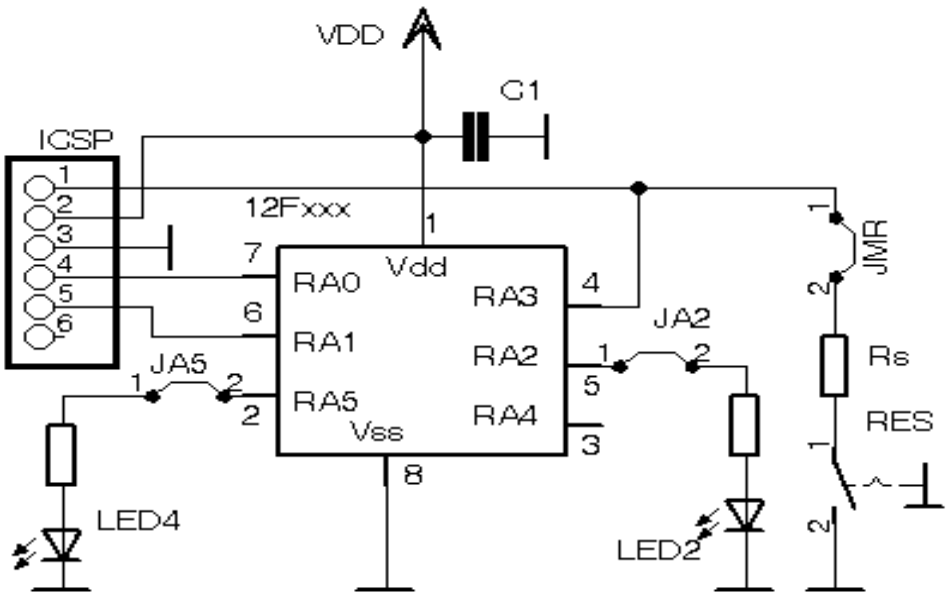
Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC12F1572).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es8C.***

8.3 - Variazione 1

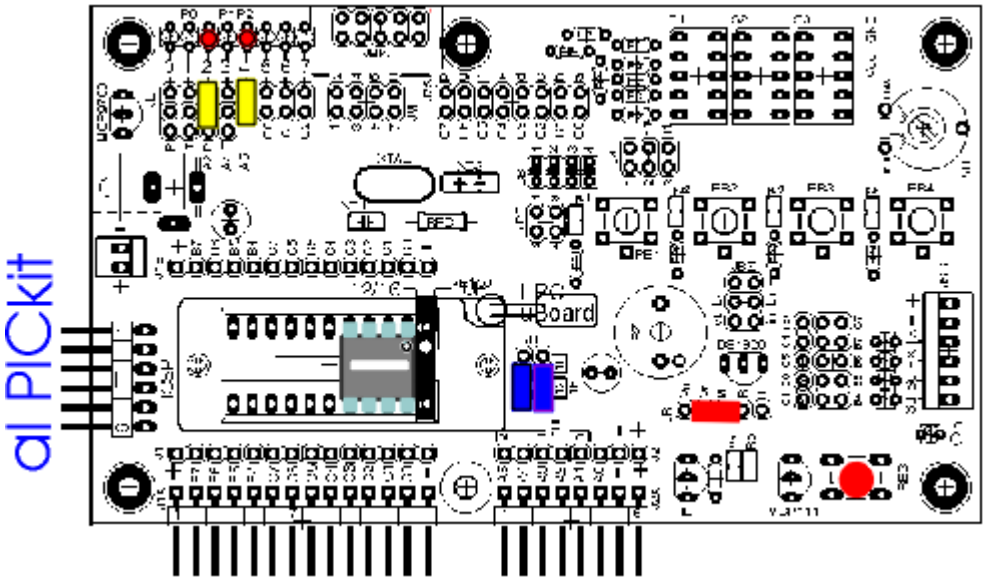
Si potrà notare che dal punto di vista dell'efficienza il programma non presenta particolari vantaggi rispetto alla soluzione dell'esercitazione 2: il processore è essenzialmente bloccato per la maggior parte del tempo in un polling sul flag di overflow e sarebbe non semplice inserire altre attività.

Però, **possiamo attivare la richiesta di interrupt del timer** e, durante l'attesa, svolgere tranquillamente altre azioni.

Le indicazioni dei componenti ricalcano quelle della scheda **LPCuB** (www).



Sulla **LPBuC** (www) la configurazione è immediata, aggiungendo un ponticello



8.5 – Interrupt

Prima di passare al sorgente, è obbligo dire due parole sull'interrupt.

Interrupt, come dice il nome, è l'interruzione del flusso delle istruzioni per svolgere momentaneamente una altro compito. L'esempio classico è quello di chi sta leggendo un libro (flusso principale) e si deve alzare per rispondere ad una telefonata (interruzione) per poi tornare alla lettura.

Interrupt è una delle principali e indispensabili funzioni dell'hardware del microcontroller, che permette l'esecuzione di più task nello stesso programma.

Le richieste di interruzione arrivano dalle periferiche integrate o da segnali esterni applicati ad alcuni pin. Si tratta di segnalare alla MPU un evento che richiede attenzione immediata.
L'interruzione richiede al processore di interrompere l'esecuzione del programma corrente e di eseguire un codice a partire da una locazione specifica (vettore di interrupt).
Questo si ottiene sostituendo il contenuto del Program Counter con l'indirizzo del vettore di interrupt.
A questo indirizzo dovrà essere scritta la routine di gestione dell'evento, al termine della quale verrà ripristinato il Program Counter al punto del programma in cui era avvenuta l'interruzione.

Si tratta di un processo analoga alla chiamata di una subroutine, che, eseguita, riporta il Program Counter all'istruzione successiva alla chiamata, ma con alcune differenze:

- una subroutine è sincrona, ovvero è chiamata volontariamente ad un certo punto del programma;
  l'interrupt è asincrono: può essere chiamato in qualsiasi momento dato che la richiesta non arriva da una istruizone del programma, ma è generata da una periferica

- il codice della subroutine occupa in memoria una serie di locazioni che dipendono dalla compilazione;
  il codice dell'interrupt inizia obbligatoriamente da una locazione fissa pre definita (vettore di interrupt)

Nel caso dei PIC Enhanced, la locazione del vettore di interrupt è a **`0004h`** nella memoria programma.
PIC con due livelli di priorità di interrupt hanno due vettori di interrupt (**`0004h`** e **`0008h`**), come i PIC18F, che, però, negli ultimi modelli hanno una tabella per multi livelli di interruzione, come i PIC di categorie superiori.

Altre differenze sono:

- la **subroutine** viene chiamata in un momento specifico e svolge le azioni richieste dal programma principale. Essendo avviata volontariamente ad un dato punto del programma, il programmatore sa quali parametri andranno in ingresso ed uscita

- **l'interruzione** può capitare in qualsiasi momento dell'esecuzione del programma principale e questo richiede che che vengano salvati obbligatoriamente alcuni registri chiave (core

register) che possono essere mdificati durante l'esecuzione della routine di interruzione e che vanno poi ripristinati al rientro nel flusso principale per riprendere l'esecuzione in modo corretto al punto in cui era stata sospesa.

Questa azione di ***save & restore*** nei Midrange richiede una serie di istruzioni in ingresso e uscita dalla gestione dell'interruzione, con un certo impiego di memoria programma, RAM e tempo.

Negli Enhanced save e restore sono del tutto automatici e non richiedono alcun intervento da parte del programmatore, non impegnano tempo e utilizzano locazioni RAM dedicate.

Un sensibile vantaggio.

L'automatismo degli Enhanced salva i registri **`STATUS, WREG, BSR, PCLATH, FSR0, FSR1`** in una area detta ***Shadow Registers***, che, comunque, è accessibile da programma.

Sia la subroutine, sia l'interrupt , **salvano nello stack l'indirizzo di rientro**. Se il numero di chiamate è alto, è possibile che lo stack non sia più sufficiente e il compilatore C va ad utilizzare stack software che richiedono risorse e tempo di esecuzione.
Per questa ragione lo stack a 16 livelli degli Enhanced offre deciso un miglioramento nelle prestazioni rispetto agli stack a 8 e 2 livelli dei PIC inferiori.

La filosofia operativa delle interruzioni nei PIC è la seguente:

- ogni periferica che può essere causa di interruzione è dotata di
  - un bit **IF** (interrup Flag) che **segnala l'avvenuto evento**
  - un bit **IE** (interrupt Enable) che **abilita la richiesta di interruzione**

Al momento dell'evento (es. overflow del timer, arrivo di un segnale esterno, fine di una conversione AD, ecc.) il flag **IF** viene settato, indipendentemente dallo stato del bit **IE.**

- Se è stato settato da programma anche lo switch IE, l'evento può avere un effetto esterno (ad esempio, nel caso di IOC, permette l'uscita dalla condizione di sleep).
- Se è stato settato anche lo switch generale di abilitazione delle interruzioni, verrà generato un interrupt

Da osservare che Microchip classifica le **sorgenti di interruzione** in due classi:
- **non periferiche**, che comprendono INT, Timer0 e IOC
- **periferiche**, tutte le altre

La differenza consiste nel fatto che l'interruzione è abilitata:
- per le **non periferiche , dal solo bit GIE**
- per le **periferiche dal bit GIE e dal bit PEIE** contemporaneamente settati

Sono sorgenti non periferiche il **Timer0**, il pin di interrupt esterno **`INT`**, la funzione **`IOC`**.
Sono sorgenti periferiche tutte le altre.

I bit di flag e di controllo delle sorgenti non periferiche sono contenuti nel registro **INTCON**.
I bit e i flag di controllo delle sorgenti periferiche sono contenuti in più registri **PIEx** (per i bit **IE**) e **PIRx** (per i bit **IF**).

Altra importante caratteristica è che, a parte poche eccezioni, i flag IF, una volta settati, restano così fino a che non sono resettati da programma. Questo serve al programma per non perdere un evento: il flag sarà azzerato dopo il completamento delle attività richieste dall'interruzione.

Se non si cancella il flag, il risultato è quello di non poter individuare successivi eventi in polling e, in interrupt, all'impossibilità di uscire dalla routine di interruzione.

Più precisamente:

- il flag **IF** a 1 segnala la causa dell'interruzione ed avvia la richiesta di interruzione
- il programma esegue la routine connessa con l'interruzione e cancella il flag **IF**.
  Se non lo fa, appena si esce dalla gestione dell'interrupt, il flag **IF** a 1 richiama immediatamente un rientro nella gestione, nella quale si resta bloccati.

Anche se usiamo un polling per testate l'evento sul flag IF, occorrerà cancellare questo flag per poter rilevare il prossimo evento.

E' necessario anche aver cura di cancellare il flag IF di una sorgente di interrupt prima di settare il bit IE che ne abilita l'interruzione.

Questo si rende necessario perchè dopo il POR o un reset generico, la situazione dei flag IF potrebbe non essere definita a 0 e questo avvierebbe richieste di interruzione indesiderate.

Altre informazioni sull'interrupt le trovate qui (www).

Vediamo come operare con l'interrupt del Timer0.

8.5 – Il programma con interrupt

Avendo a che fare con interrupt, è necessario averne ben chiaro il funzionamento e definire bene le azioni del programma principale e quelle richieste dall'interruzione.

Nel nostro caso vogliamo sovrapporre la gestione del LED lampeggiante, che avviene attraverso un interrupt, con quella di un pulsante che accende e spegne un altro LED, gestita dal main.

Pare ovvio attribuire all'interruzione il lampeggio del LED: in sostanza, si tratta di intervenire a cadenza di tempo fissa per cambiarne lo stato.

La sequenza necessaria a comandare il LED dal pulsante l'abbiamo già vista nell'esercitazione 3:

```
#define Pulsante PORTAbits.PORTA3
#define Led1     LATAbits.LATA5     // LED dipendente dal pulsante
#define Led2     LATAbits.LATA2     // LED lampeggiante

/************************ MAIN PROGRAM ***************************/
void main(void){

    // Inizializza I/O
    LATAbits.LATA5 = 1;          // preset latch RA5 =1
    TRISA = 0b11011111;          // RA5 come output

    // abilitiamo wpu
    OPTION_REGbits,nWPUEN = 0;

    while (1){                   // loop continuo
       while (Pulsante==1);      // attesa pressione pulsante
       Led1=~Led1;               // toggle LED
       __delay_ms(30);           // debounce 30ms

       while (Pulsante==0);      // attesa pulsante aperto
       __delay_ms(30);           // debounce 30ms
    }
}
```

In questa base andiamo ad innestare l'interrupt.

Settiamo il Timer0 come visto prima

```
// Timer0 Registers: Prescaler=1:256; TMR0 Preset=12; Period=499,712 ms
   OPTION_REGbits.TMR0CS = 0;   // Clock Source = Internal Clock
   OPTION_REGbits.PSA  = 0;     // bit 3 Prescaler to Timer0
   OPTION_REGbits.PS  = 0b111;  // PS2:PS0: Prescaler 1:256
   TMR0 = 10;                   // preset for counter
```

Ora aggiungiamo la routine di risposta all'evento di interrupt:

```
/**** Interrupt Routine ****/
void interrupt irq(){        // C90
//void __interrupt() irq(void){  // C99

    TMR0 = 10 ;              // ricarica counter
```

```
    INTCONbits.TMR0IF = 0;   // cancella flag
    Led2=~Led2;              // inverti LED
}
```

La routine di gestione dell'interruzione è al di fuori del loop del main: non è una funzione chiamata dal programma, ma è chiamata da un evento hardware.
Uno scambio di dati tra main e interrupt può essere comunque effettuato usando delle variabili global che in questo modo saranno accessibili da tutte le funzioni. In aggiunta, queste locazioni di RAM modificate nella routine di interruzione, potranno utilmente essere dichiarate volatile per evitare che il compilatore segnali ridondanze non esistenti.

Per la routine di interruzione si può utilizzare la forma:

- **`interrupt`** che è compatibile con C90 o
- **`__interrupt`** che è compatibile con C99

Il nome attribuito alla funzione (qui è **`irq`**) non è importante e può essere variato a piacere.

La forma compatibile con C99 richiede obbligatoriamente:

- una doppia sottolineatura prima di **`interrupt`**
- una coppia di parentesi dopo **`interrupt`**

Questa coppia di parentesi, apparentemente "inutile", serve a contenere le informazioni per quelli che dispongono di più livelli; è vuota nel caso di processori con un solo livello di interruzione.
Il filmato *New Interrupt Syntax in MPLAB® XC8 Webinar* (www) può essere utile per chiarire la situazione dell'interrupt per i processori a 8 bit.

Nel testo riportiamo entrambe le forme, di cui quella C99 è come commento. Volendo passare a questa, basta decommentare la linea e mettere a commento la versione C90.

Le due forme non sono compatibili una con l'altra. Ad esempio, se utilizziamo la forma C99 con una vecchia versione del compilatore (XC8 v1.35) otteniamo un warning:

```
warning: (1429) attribute "interrupt" is not understood by the compiler; this attribute will be
ignored
```

Per contro, se utilizziamo la forma C90 con la versione 2. del compilatore, otteniamo non uno, ma ben due segnalazioni di errore:

```
../../es8c_1.c:98:6: error: variable has incomplete type 'void'
void interrupt isr(void){
     ^
../../es8c_1.c:98:15: error: expected ';' after top level declarator
void interrupt isr(void){
```

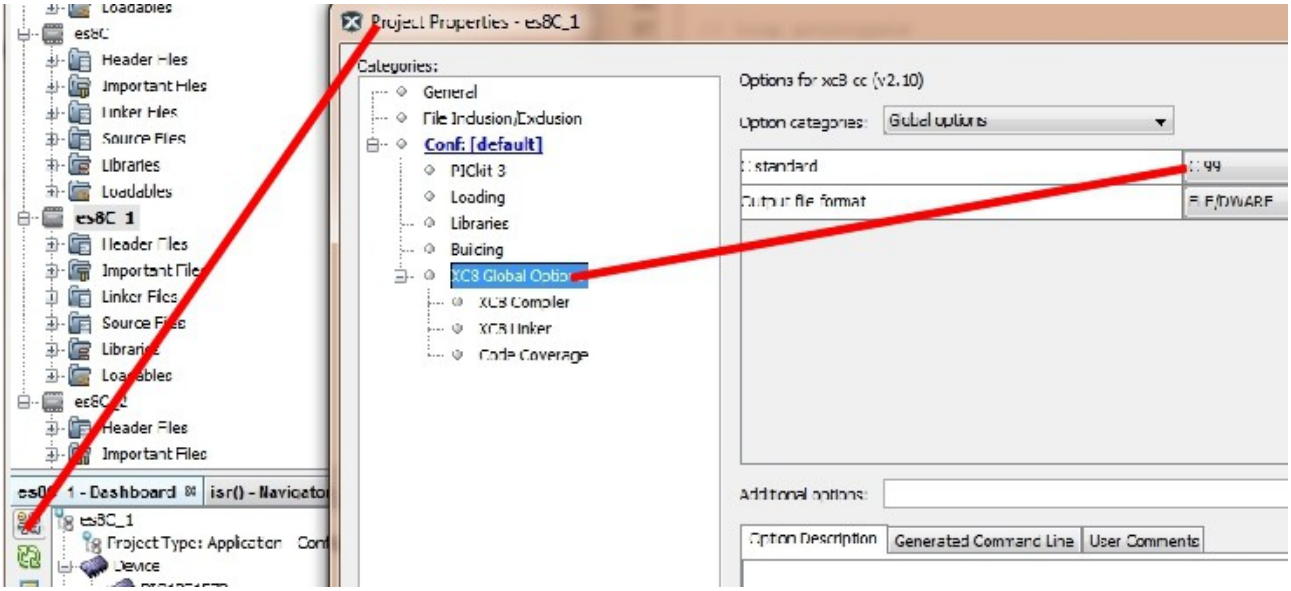
delle quali la seconda si riferisce comunque alla stessa riga.

In particolare, versioni del compilatore XC8 successive alla 2.0 partono per default con la compatibilità C99: aprendo un progetto esistente, funzionante con compilatori di versioni precedenti, si potranno avere errori dovuti alla diversa forma richiesta.

Sono possibili due azioni:

- modificare il sorgente adattando la sintassi richiesta
- commutare nelle caratteristiche globali del progetto la compatibilità XC8 da C99 a C90.

Volendo effettuare il cambio dello standard, occorre:



- accedere dalla ***Dashboard*** del progetto alle ***Project Properties*** -> **XC8 Global Options**
- selezionare ***C90*** nella casella ***C Standard***

La selezione vale solamente per il progetto in corso

Abbiamo anche bisogno di un contatore per gli eventi, per cui dichiariamo una variabile char (8 bit senza segno) che inizializziamo a 0. La dichiariamo di tipo static per allocarla ad un indirizzo fisso di RAM.

Nel main occorrerà abilitare l'interrupt del timer e quello generale:

```
 TMR0IF=0 ;                      // clear flag
// abilita interrupt Timer0 e generale
 INTCONbits.TMR0IE = 1;
 GIE = 1;                       // ei();
```

Basta abilitare `GIE` , dato che il Timer0 è una sorgente non periferica e quindi non richiede anche **`PEIE`**.
Esiste una macro **`ei()`** che equivale a **`GIE=1`**, ma non sembra proprio il caso di utilizzarla perchè il risparmio di caratteri sarebbe trascurabile contro una non immediata comprensione.

Osserviamo che prima di abilitare la sorgente dell'interruzione, ne abbiamo cancellato il flag IF per evitare una immediata entrata nella gestione dell'interrupt se il flag fosse a 1: il timer inizia a contare appena riceve alimentazione e clock e potrebbe aver raggiunto un overflow prima dell'abilitazione dell'interruzione.

Ora il LED1 lampeggerà una volta al secondo e il LED2 sarà comandato dal pulsante.

Se non è ancora chiaro il meccanismo, quello che succede è quanto segue:

- il programma ha pre caricato il timer e attivato l'interrupt
- il programma segue in polling lo stato del pulsante e comanda il LED2 di conseguenza
- ogni 499ms il programma viene interrotto, ovunque si trovi nella lista delle istruzioni, e viene esguita la gestione dell'interruzione, che provvede a cambiare stato al LED1
- alla fine della gestione dell'interruzione, si torna ala programma principale che riprende nell'esatto punto in cui era stato interrotto

Osserviamo che non c'è alcuna azione di save e restore dei registri del core, in quanto, negli Enhanced, come abbiamo detto, questo avviene automaticamente.

La routine di gestione dell'interrupt è estremamente semplice, dato che esiste una sola sorgente di interruzione attiva e contiene tutte le istruzioni necessarie a gestire l'evento.
La situazione però non è certo sempre così semplice, sopratutto quando si ha a che fare con più sorgenti di interrupt attivate e con periferiche complesse da gestire.

 In generale, **la via migliore è quella di mantenere ridotta al minimo indispensabile la routine** e passare i dati al main per l'elaborazione.

es8C_1

```
/*****************************************************************
*-----------------------------------------------------------------
*    Titolo        :  C Enhanced - es8C_1
*                     Un LED collegato tra R5 e Vss lampeggia una
*                     volta al secondo con Timer0 mentre un pulsante
*                     comanda un altro LED.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  12F1571/2
*    Autore        :  afg
*
*-----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ---------------
*       12F1571/2 @ 8 pin
*
*                   |‾‾\/‾‾|
*              Vdd -|1    8|- Vss
*              RA5 -|2    7|- RA0
*              RA4 -|3    6|- RA1
*       RA3/MCLR -|4    5|- RA2
*                   |______|
*
*    Vdd               1: ++
*    RA5/OSC1/CLKIN    2: Out LED
*    RA4/OSC2/CLKOUT   3: In
*    RA3/!MCLR/VPP     4: MCLR
*    RA2               5: In
*    RA1/ICSPCLK       6: In
*    RA0/ICSPDAT       7: In
*    Vss               8: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/

// intosc, no clockout, no wdt, no pwrt, no mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1572_1.h"

#include <xc.h>

#define Led4 LATAbits.LATA5     // LED comandato dal pulsante
#define Led2 LATAbits.LATA2     // LED lampeggiante
#define Pulsante PORTAbits.RA3
#define _XTAL_FREQ   500000

/************************ MAIN PROGRAM ****************************/
void main(){

    // Inizializza I/O
    ANSELA=0;               // no analogiche
    LATAbits.LATA5 = 1;     // preset Led4 on
    LATAbits.LATA2 = 0;     // preset Led2 off
    TRISAbits.TRISA5 = 0;   // RA5 come output
    TRISAbits.TRISA2 = 0;   // RA2 come output
```

```
  // abilitiamo wpu per protezione pin non usati
   OPTION_REGbits.nWPUEN = 0;

  // Timer0 Registers: Prescaler=1:256; TMR0 Preset=12; Period=499,712 ms
   OPTION_REGbits.TMR0CS = 0;   // Clock Source = Internal Clock (CLKO)
   OPTION_REGbits.PSA  = 0;     // bit 3 Prescaler to Timer0
   OPTION_REGbits.PS  = 0b111;  // PS2:PS0: Prescaler 1:256
   TMR0 = 10;                   // preset for counter
   TMR0IF=0 ;                   // clear flag

  // abilita interrupt Timer0 e generale
   INTCONbits.TMR0IE = 1;
   GIE = 1;                     // ei();

// Loop principale
    while (1){                   // loop continuo
       while (Pulsante==1);      // attesa pressione pulsante
       Led4=~Led4;               // toggle LED
       __delay_ms(30);           // debounce 30ms

       while (Pulsante==0);      // attesa pulsante aperto
       __delay_ms(30);           // debounce 30ms
    }
}
/**** Interrupt Routine ****/
void interrupt isr(){       // C90
// void __interrupt() isr(void){  // C99
    TMR0 = 10 ;             // ricarica counter
    INTCONbits.TMR0IF = 0;  // cancella flag
    Led2=~Led2;             // inverti LED
}
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC12F1572).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es8C_1.***

8. 6 - Variazione 2

Possiamo trovarci nella necessità di realizzare cadenze di tempo più lunghe di quanto il contatore del timer possa permettere.
Schema e disposizione sulla **LPCuB** (www) sono identici all'esempio precedente.

Ad esempio, se il clock è 8MHz, il massimo tempo realizzabile è di 32.768ms. Non sarebbe possibile ottenere i 500ms necessari a far lampeggiare il LED.

Certamente diventa possibile con un semplice algoritmo:

- generiamo un interrupt con un periodo minore del massimo ottenibile, ad esempio 2ms
- ad ogni interruzione, avanziamo un contatore e , totalizzati 250 eventi, avremo raggiunto i 500ms richiesti

La modifica della routine di interrupt è rapida:

```
/**** Interrupt Routine ****/
void interrupt irq(){
static unsigned char irqcntr=0;
    TMR0 = 10 ;              // ricarica counter
    INTCONbits.TMR0IF = 0;   // cancella flag
    ++irqcntr1;
    if (irqcntr==250){
     irqcntr=0;              // azzera contatore
    Led2=~Led2;              // inverti LED
    }
}
```

Per la variabile **irqcntr** utilizziamo sempre un **char** (8bit) dato che il massimo valore conteggiato è inferiore a 255; di conseguenza non è necessario un **int** a 16bit.

es8C_2

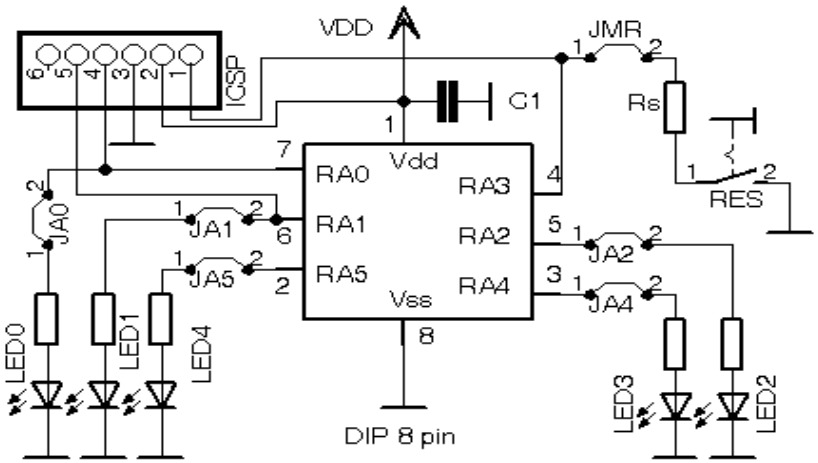
```
/*****************************************************************
*-----------------------------------------------------------------
*    Titolo        :  C Enhanced - es8C_2
*                     Un LED collegato tra R5 e Vss lampeggia una
*                     volta al secondo con Timer0 mentre un pulsante
*                     comanda un altro LED.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  12F1571/2
*    Autore        :  afg
*
*-----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       12F1571/2 @ 8 pin
*
*                   |‾‾\/‾‾|
*             Vdd -|1    8|- Vss
*             RA5 -|2    7|- RA0
*             RA4 -|3    6|- RA1
*        RA3/MCLR -|4    5|- RA2
*                   |______|
*
*    Vdd               1: ++
*    RA5/OSC1/CLKIN    2: Out LED
*    RA4/OSC2/CLKOUT   3: In
*    RA3/!MCLR/VPP     4: MCLR
*    RA2               5: In
*    RA1/ICSPCLK       6: In
*    RA0/ICSPDAT       7: In
*    Vss               8: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/

// intosc, no clockout, no wdt, no pwrt, no mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1572_1.h"

#include <xc.h>

#define Led1 LATAbits.LATA5
#define Led2 LATAbits.LATA2
#define Pulsante PORTAbits.RA3
#define _XTAL_FREQ   8000000

/************************ MAIN PROGRAM ****************************/
void main(){

    OSCCON = 0b01110000;    // clock a 8MHz
    // Inizializza I/O
    ANSELA=0;               // no analogiche
    LATAbits.LATA5 = 1;     // preset Led1 on
    LATAbits.LATA2 = 0;     // preset Led2 off
```

```
    TRISAbits.TRISA5 = 0;   // RA5 come output
    TRISAbits.TRISA2 = 0;   // RA2 come output

  // abilitiamo wpu per protezione pin non usati
   OPTION_REGbits.nWPUEN = 0;

  // Timer0 Prescaler=1:256; TMR0 Preset=4; Period=2.00 ms
   OPTION_REGbits.TMR0CS= 0;    // Clock Source = Internal Clock
   OPTION_REGbits.PSA = 0;      // bit 3 Prescaler to Timer0
   OPTION_REGbits.PS = 0b011;   // PS2:PS0: Prescaler 1:16
   TMR0 = 4;                    // preset for counter
   TMR0IF=0 ;                   // clear flag

  // abilita interrupt Timer0 e generale
   INTCONbits.TMR0IE = 1;
   GIE = 1;                     // ei();

// Loop principale
    while (1){                   // loop continuo
       while (Pulsante==1);      // attesa pressione pulsante
       Led1=~Led1;               // toggle LED
       __delay_ms(30);           // debounce 30ms

       while (Pulsante==0);      // attesa pulsante aperto
       __delay_ms(30);           // debounce 30ms
    }
}
/**** Interrupt Routine ****/
void interrupt isr(){       // C90
// void __interrupt() isr(void){  // C99

static unsigned char irqcntr=0;

    TMR0 = 4 ;                   // ricarica counter
    INTCONbits.TMR0IF = 0;       // cancella flag

    ++irqcntr;                   // incrementa counter

    if (irqcntr==250){           // se 250 conteggi da 2ms
     irqcntr=0;                  // azzera contatore

    Led2=~Led2;                  // inverti LED
    }
}
```

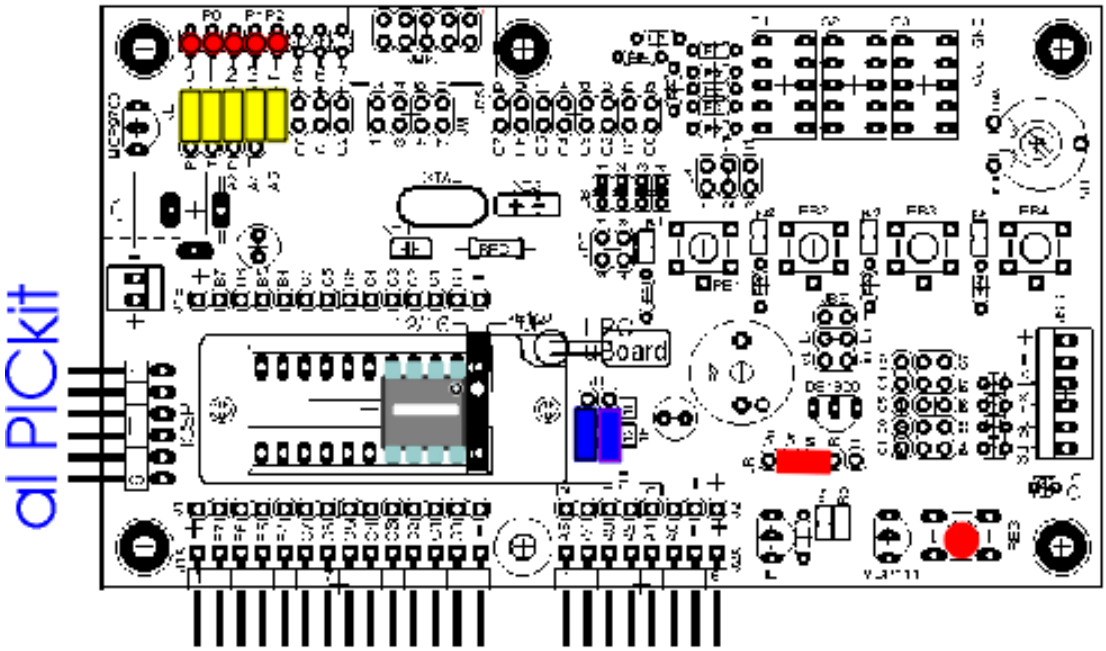
Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC12F1572).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es8C_2.***

8. 7 - Variazione 3

In base a quanto visto sopra possiamo pensare di usare l'interrupt del timer come temporizzazioni per operazioni non sincrone.
A quanto visto al punto precedente aggiungiamo altri tre LED che devono lampeggiare con periodi diversi. Le indicazioni sullo schema ricalcano la situazione sulla scheda di sviluppo **LPCuB** (www)



Come al solito sulla **LPCuB** (www) è solo questione di jumper:



Da notare: fino ad ora non abbiamo mai coinvolto i pin **RA0** e **RA1** nelle esercitazione. La ragione è semplice: questi pin costituiscono l'interfaccia di programmazione del chip, accessibile dal connettore **ICSP**. Durante la programmazione (e il debug) su questi pin si svolge una comunicazione seriale sincrona tra tools e chip, che non deve essere disturbata dall'applicazione di carichi che possano deformare i fronti di commutazione dei segnali.

Nel caso della scheda LPCuB (www), chiudendo il ponticello **AB0** e **AB1T** si collega al pin **RA1** il **LED 1** e al pin **RA0** il LED0; però, questi LED assorbono una corrente minima (1-2mA) e non creano alcun problema durante la programmazione. Semplicemente, li si vedrà lampeggiare molto velocemente durante questa fase, dato che le uscite dei tools di Microchip sono ben bufferate e non risento del piccolo carico.
Ben diversa sarà la situazione se ai due pin abbiamo collegato carichi maggiori, sopratutto se capacitivi. In tal caso jumper per separare questi carichi durante la programmazione on board sono indispensabili.

 Se usate altre basi di sviluppo, è molto opportuno NON avere carichi collegati a RA0/1 durante la programmazione.

Utilizziamo, come prima,
- il LED4, collegato a **RA5**, per essere comandato dal pulsante su RA3.
- Il LED2 , collegato a **RA2**, con un lampeggio a 500ms

A questi aggiungiamo:
- il LED1, collegato a **RA1**, con un lampeggio a 380ms
- il LED3, collegato a **RA4**, con un lampeggio di 170ms
- il LED0, collegato a **RA0**, con un lampeggio di 800ms

Come in precedenza, **MCLR** viene sostituito nel config con **RA3** e il pulsante RES non necessita di pull-up, dato che viene abilitato quello integrato.

Sempre con un clock a 4MHz possiamo generare un interrupt con un periodo di 10ms, maggiore rispetto a quello visto nella variazione precedente. In generale, dove possibile, è più opportuno scegliere una cadenza delle interruzione non troppo rapida, per evitare che una eccessiva frequenza delle interruzioni impedisca un corretto svolgimento delle azioni del main.

Con 10ms, sarà necessario contare 17 overflow (17x10ms=170ms) per il LED più veloce, 38 e 50 per quelli intermedi e 80 per quello più lento. Quindi, definiamo altri tre contatori:

La modifica della routine di interrupt è rapida: vengono aggiunti i contatori necessari e le selezioni **if** relative

```
/**** Interrupt Routine ****/
void interrupt isr(){       // C90
// void __interrupt() isr(void){  // C99

static unsigned char irqcntr1=0;
static unsigned char irqcntr2=0;
static unsigned char irqcntr3=0;
static unsigned char irqcntr4=0;

    TMR0 = 98 ;              // ricarica counter
    INTCONbits.TMR0IF = 0;   // cancella flag
    ++irqcntr1;
```

```
    ++irqcntr2;
    ++irqcntr3;
    ++irqcntr4;

    if (irqcntr1==50){
     irqcntr1=0;              // azzera contatore
    Led2=~Led2;               // inverti LED
    }
    if (irqcntr2==38){
     irqcntr2=0;              // azzera contatore
    Led3=~Led3;               // inverti LED
    }
    if (irqcntr3==17){
     irqcntr3=0;              // azzera contatore
    Led1=~Led1;               // inverti LED
    }
    if (irqcntr4==80){
     irqcntr4=0;              // azzera contatore
    Led0=~Led0;               // inverti LED
    }
}
```

Le condizioni `if`, eseguite in successione, azzerano il relativo contatore e commutano il LED al raggiungimento del numero di conteggi totalizzati.

Questa può essere una tecnica per utilizzare un solo timer come per task con periodi diversi: il timer fornisce una base tempi costante, che le task utilizzano per le loro temporizzazioni. Nell'esempio le task sono costituite solamente dall'inversione dello stato di un LED, ma, ovviamente, potranno effettuare altre operazioni.
Da notare che lo svolgimento delle istruzioni della routine di interrupt dovrà essere molto minore dell'intervallo tra una interruzione e la successiva, per permettere al main di svolgere il suo lavoro e, sopratutto, non sovrapporsi con la precedente. Nel caso dell'esempio, con una cadenza di 10ms, la durata dell'esecuzione della routine di interruzione dovrà essere molto minore di 10ms.
La cadenza dell'interruzione dovrà essere proporzionata alle necessità dell'applicazione

es8C_3

```
/*****************************************************************
*----------------------------------------------------------------
*    Titolo        :  C Enhanced - es8C_3
*                     Quattro LED lampeggiano a frequenze diverse con
*                     Timer0 mentre un pulsante comanda un altro LED.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  12F1571/2
*    Autore        :  afg
*
*----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       12F1571/2 @ 8 pin
*
*                   |__\/__|
*             Vdd -|1    8|- Vss
*             RA5 -|2    7|- RA0
*             RA4 -|3    6|- RA1
*        RA3/MCLR -|4    5|- RA2
*                   |______|
*
*    Vdd               1: ++
*    RA5/OSC1/CLKIN    2: Out LED4
*    RA4/OSC2/CLKOUT   3: Out LED3
*    RA3/!MCLR/VPP     4: In  Pulsante
*    RA2               5: Out LED2
*    RA1/ICSPCLK       6: Out LED1
*    RA0/ICSPDAT       7: Out LED0
*    Vss               8: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/

// intosc, no clockout, no wdt, no pwrt, no mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1572_1.h"

#include <xc.h>

#define Led0 LATAbits.LATA0
#define Led1 LATAbits.LATA1
#define Led2 LATAbits.LATA2
#define LED3 LATAbits.LATA4
#define LED4 LATAbits.LATA5
#define Pulsante PORTAbits.RA3

#define _XTAL_FREQ   4000000

/************************ MAIN PROGRAM ***************************/
void main(){

    OSCCON = 0b01101000;    // clock a 4MHz
    // Inizializza I/O
```

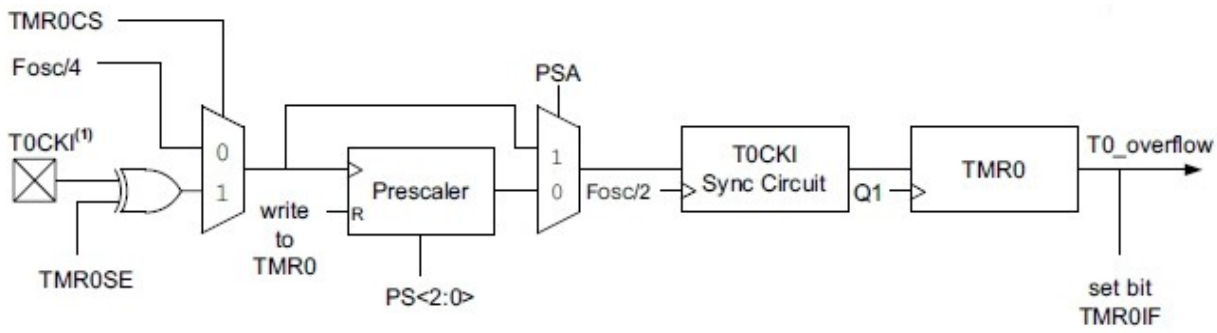
```
    ANSELA=0;               // no analogiche
    LATAbits.LATA5 = 0;     // preset Led4 off
    LATAbits.LATA0 = 1;     // preset Led0 on
    LATABITS.LATA1 = 0;     // preset Led1 on
    LATAbits.LATA2 = 0;     // preset Led2 off
    LATAbits.LATA3 = 1;     // preset Led3 on
    TRISA =0b110001001;     // RA5/4/2/1 come output

  // abilitiamo wpu per RA3
   OPTION_REGbits.nWPUEN = 0;

  // Timer0 Prescaler=1:64; TMR0 Preset=98; Period=10.00 ms
   OPTION_REGbits.TMR0CS= 0;    // Clock Source = Internal Clock
   OPTION_REGbits.PSA = 0;      // bit 3 Prescaler to Timer0
   OPTION_REGbits.PS = 0b101;   // PS2:PS0: Prescaler 1:64
   TMR0 = 98;                   // preset for counter
   TMR0IF=0 ;                   // clear flag

  // abilita interrupt Timer0 e generale
   INTCONbits.TMR0IE = 1;
   GIE = 1;                     // ei();

// Loop principale
    while (1){                  // loop continuo
       while (Pulsante==1);     // attesa pressione pulsante
       Led1=~Led1;              // toggle LED
       __delay_ms(30);          // debounce 30ms

       while (Pulsante==0);     // attesa pulsante aperto
       __delay_ms(30);          // debounce 30ms
    }
}
/**** Interrupt Routine ****/
void interrupt isr(){       // C90
// void __interrupt() isr(void){  // C99

static unsigned char irqcntr1=0;
static unsigned char irqcntr2=0;
static unsigned char irqcntr3=0;
static unsigned char irqcntr4=0;

    TMR0 = 98 ;             // ricarica counter
    INTCONbits.TMR0IF = 0;  // cancella flag
    ++irqcntr1;             // incrementa contatori
    ++irqcntr2;
    ++irqcntr3;
    ++irqcntr4;

    if (irqcntr1==50){      // 50x10ms=500ms
     irqcntr1=0;            // azzera contatore
    Led2=~Led2;             // inverti LED
    }
    if (irqcntr2==38){      // 38x10ms=380ms
     irqcntr2=0;            // azzera contatore
    Led3=~Led3;             // inverti LED
    }
    if (irqcntr3==17){      // 17x10ms=170ms
     irqcntr3=0;            // azzera contatore
    Led1=~Led1;             // inverti LED
```

```
    }
    if (irqcntr4==80){      // 80x10ms=800ms
     irqcntr4=0;            // azzera contatore
    Led0=~Led0;             // inverti LED
    }
}
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC12F1572).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es8C_3*.**

8. 8 – Altre possibilità del Timer0

Se osserviamo il diagramma funzionale del **Timer0** offerto dal foglio dati, abbiamo sott'occhio una rapida idea delle possibilità:



Il Timer0 può ricevere il clock da una sorgente esterna attraverso il pin **`T0CKI.`** Qui si può iniettare un segnale con una **frequenza massima di FOSC/8.**
Questo segnale alimenta il contatore se il bit **`TMR0CS`** è a 1, mentre il bit **`TMR0SE`** seleziona il fronte del segnale su cui avviene il conteggio.
Quando si utilizza il clock esterno questo è sincronizzato con il ciclo istruzione.
Se non si applica il prescaler, **`TMR0`** può essere usato come contatore di impulsi esetrni fino a 255.

Il prescaler, che può essere bypassato se **`PSA`** è a 1, divide sia il segnale del clock interno, sia quello del clock esterno.

In relazione ai "clock" può essere utile chiarire che:

- **`_XTAL_FREQ`** o **`FOSC`** è la frequenza dell'oscillatore
- **`FOSC/4`** o **`_XTAL_FREQ/4`** è la frequenza di cilo delle istruzioni, pari a 1/4 di quella dell'oscillatore
- **`tcyc`** o **`1/FOSC/4`** è il periodo di una istruzione

Quindi, ad esempio:

`_XTAL_FREQ`	4MHz
`FOSC/4`	1MHz
`tcyc`	1us

In modalità sleep il Timer0 non funziona, anche se alimentato da un clock esterno, ma il contenuto di **`TMR0`** è conservato.
Questo registro può essere letto e scritto, ma la sua lettura raramente è significativa in quanto se viene usato un prescaler, il contenuto di questo non è accessibile e se non si usa il prescaler, con il clock interno il rate di avanzamento è del conteggio è pari a quello delle istruzioni.

L'uscita di **`TMR0`** (bit **`TMR0IF`**) può essere usata come gate del **Timer1**.

Inoltre, esistono altre variazioni del Timer0, analoghe a quella disponibile nei PIC18F, con registro di conteggio a 16bit, il che permette una maggiore ampiezza/definizione dei periodi ottenibili.

Schede informative dettagliate. Esercitazione 8

Qui di seguito alcune pagine di informazioni più dettagliate sua argomenti trattati nella prima esercitazione. Che possono essere estratte e conservate separatamente.

- **I timer nei PIC**

Files allegati

Sono allegati i seguenti files/cartelle:

- **Cartella *es8C* : progetti e sorgenti**

I timer nei PIC.

Il modulo timer è parte integrante di qualsiasi famiglia di microcontrollori.
La maggior parte dei PIC moderni dispone di più di un timer con funzioni generiche di timer/counter, più altri timer pensati principalmente come supporto di specifiche attività, come il modulo CCP/PWM.
In pratica, gran parte delle applicazioni ha bisogno di un timer e la presenza di più moduli permette di affrontare situazioni anche complesse.

In generale, il timer derivare il suo clock da quello principale, ovvero dal clock delle istruzioni: in questo caso si dice che funzioni come timer ed esegue principalmente operazioni di generazioni di intervalli di tempo.
Il timer, però, può essere realizzato per ricevere un clock dall'esterno del microcontroller e in tal caso si dice che operi come counter, ad esempio per misurazioni di tempo.
Il programmatore può utilizzare queste funzioni in base alle necessità dell'applicazione. La precisione dei tempi e delle misure dipende da quella del clock che alimenta il timer.

Nei PIC il contatore è costituito da un registro a 8 o a 16 bit, che conta in salita a partire da 0 fino ad un massimo di 256 (8 bit) o 65535 (16 bit).
Quando il conteggio ha raggiunto il massimo, si verifica un overflow e viene settato un flag che indica l'evento. L'abilitazione di uno switch specifico consente di generare una interruzione (interrupt).

Come standard, si ha la seguente situazione all'overflow:

- in ogni caso viene settato il bit **`IF`** relativo al timer
- se è stato settato il bit di abilitazione **`IE`** e gli switch **`GIE/PEIE`**, all'overflow fa seguito una richiesta di interruzione

Da notare che:

- **il bit IF è settato indipendentemente dall'abilitazione di IE e GIE/PEIE**
- **il bit IF va cancellato da programma** dopo aver eseguito la routine richiesta all'overflow, sia che si operi in interrupt, sia che si operi in polling.

Se il flag IF non viene cancellato da programma, permane a livello 1 e impedisce di identificare successivi eventi; se è abilitato l'interrupt, impedisce di rientrare al programma principale.

Gli impulsi di clock conteggiati dai timer sono calcolabili come:

$$N = valore\_registro * P$$

dove ***P*** è il valore del prescaler.
Ad esempio, un timer a 8 bit alimentato con un prescaler di 32 conterà al massimo

$$N = 256 * 32 = 8192\ impulsi$$

Il periodo di tempo generato dipenderà dalla frequenza del clock

Period = (valore_registro)*(Prescaler) *(1/clock)

Nel caso dell'esempio precedente, con un clock principale di 4MHz, si ha un FOSC/4=1MHz, con un periodo di 1us, per cui.

Period = N *(1) = 8192us

Il registro di conteggio dei timer è normalmente accessibile in lettura e scrittura: questo consente di pre caricare un valore nel registro, valore dal quale partirà il conteggio verso l'overflow. In questo modo è possibile ottenere una gamma di tempi molto ampia.
Ad esempio, in un timer a 8 bit, se lo pre carichiamo con 56, avremo l'overflow dopo 200 impulsi di clock, perchè il conteggio partirà da 50 e non da 0.

In base a quanto sopra posiamo calcolare quale tempo sarà generato da una certa configurazione.
Ad esempio, un timer a 8 bit, alimentato da un clock di 8MHz (FOSC/4 = 2MHz, periodo 0.5us), con prescaler 1:256 e pre caricato con 100, avrà l'overflow dopo:

T = (256-100) * 0.5 * 256 = 19.968 ms

Sono disponibili in rete numerose applicazioni che calcolano i parametri da fornire al timer per realizzare un determinato tempo. Utilizzando queste app si risparmierà certamente del tempo e si eviteranno errori.

Da notare che il periodo ottenibile ha una certa granulosità e può non essere con precisione quello voluto: ad esempio, con un timer a 8 bit alimentato da un clock a 4MHz, non è possibile ottenere 20ms precisi (si avrà un periodo di 19.968 o 20.096ms), mentre sarà possibile con un timer a 16bit. Va però aggiunto che la precisione del periodo non dipende solo dal conteggio del timer, ma risente anche dell'errore del clock, che potrebbe anche essere maggiore.

Si possono verificare casi in cui il valore del periodo richiesto è troppo grande per essere inserito nel registro di conteggio, nonostante il massimo prescaler inserito.
Questo significa che il periodo richiesto è troppo lungo per essere prodotto da quel timer. In questi casi, solitamente, si procede a caricare il timer con i parametri di un periodo possibile, sottomultiplo di quello voluto, ed a conteggiare gli overflow fino a totalizzare il tempo desiderato.