Steve Blackbird (TardoFreak)

PIERIN PIC18 IL PROGRAMMA DEMO: TIMER E SUPER VELOCITÀ PER IL MICRO

2 June 2013

Il programma di demo caricato sul **PIERIN PIC18**, quello che fa lampeggiare i due LED, per intenderci, all' apparenza sembra un programmino semplice. In effetti lo è ma dalla sua analisi possiamo scoprire alcune caratteristiche del microcontrollore che sono comuni anche a microcontrollori di fascia più alta e che lo rendono particolarmente versatile e adatto per realizzare prodotti affidabili.

Il progetto

Come ho accennato prima analizzare il progetto può essere l' occasione per capire come è stato pensato ma anche per esplorare alcune caratteristiche del microcontrollore. Penso che sia un buon metodo per imparare davvero come si lavora con questi oggetti.

Lo scopo

Lo scopo di questo programma di demo è quello di dare la possibilità a chi si trova il **PIERIN PIC18** in mano per la prima volta, di capire se funziona. Avendo a disposizione due pulsanti e due LED si vuole fare in modo che l' utilizzatore verifichi che i due LED si accendano e che i due pulsanti funzionino. Si è deciso di fare un progetto che funzioni in questo modo:

- Se nessuno dei due pulsanti è premuto i due LED lampeggiano alternativamente (mentre uno è acceso l' altro è spento) con un periodo di 800 ms (400 ms di accensione per ogni LED)
- Se si preme un pulsante il LED posto sopra di esso deve lampeggiare con un periodo di 400 ms (200 ms accesso e 200 ms spento) mentre l' atro LED resta spento.
- Se si premeno tutti e due i pulsanti i due LED devono lampeggiare insieme (accesi e spenti contemporaneamente) con un periodo di 400 ms (200 ms accessi e 200 ms spenti)

E' chiaro che se ci sono lampeggi il programma dovrà gestire o generare dei ritardi ed attivare e disattivare dei pin di uscita. Dovendo verificare il funzionamento dei

pulsanti il programma in qualche modo dovrà leggere il loro stato (premuto o rilasciato) ed agire di conseguenza.

Il programma a grandi linee

Il programma è stato scritto partendo dal progetto di base infilandoci dentro tutto quello che serve per fargli fare il lampeggio. Quindi d' ora in poi darò per scontato che si conosca il progetto di base e come è strutturato. Per chi si fosse sintonizzato solo ora ricordo che l' articolo che lo descrive lo si trova a **questo link**.
Orbene, la prima cosa da fare è pensare a come gestire i ritardi. Si possono gestire da caproni inserendo ritardi fissi, imprecisi e molto facili da usare ma che sconsiglio vivamente perché la CPU del micro non deve passare la maggior parte della sua vita a fare conteggi solo per perdere tempo, ma deve fare cose più interessanti. A questo punto tanto vale prendersi la briga di utilizzare i timer e magari di utilizzarlo bene.
Il sistema adottato è quello del' interrupt ciclica che viene utilizzata per implementare un timer software. In pratica si scrive il programma in modo che venga interrotto ciclicamente, ad intervalli conosciuti, da un timer. All' interno della parte della funzione che gestisce l' interrupt si manipola una variabile che si comporterà come un timer che si decrementa ogni volta e che arrivato a zero si ferma. Soluzione semplice ed elegante anche perché permette di ottenere ritardi lunghi senza "imballare" la CPU. Abbiamo quindi implementato una base tempi, nel nostro caso di 1ms, ed una variabile che si decrementa ogni ms. e con la quale possiamo fare ritardi lunghi a piacere. Più che bene per i LED visto che saranno guardati da un essere umano.

Usare il PLL

Ed ora una piccola chiosa su una caratteristica del microcontrollore.
Il **PIC18F47J53** ha una sezione del clock che è molto versatile ed interessante. Anche qui suppongo che si abbia a portata di mano il Data Sheet perché a quello farò riferimento. Colgo l' occasione per ricordare che il Data Sheet è da considerarsi come il Libro della Legge Sacra per un microcontrollista, quindi deve essere sempre disponibile per qualsiasi consultazione.
Stavo dicendo che la sezione del clock del micro è già una di quelle furbe. Oltre ad avere tante opzioni utili ha un PLL, un oscillatore che partendo da una frequenza ne sintetizza una che è un multiplo della frequenza d' ingresso. Utilizzare un PLL è un' ottima cosa perché permette di evitare di dovere utilizzare quarzi a frequenze assurde, che potrebbero generare emissioni RF sgradevoli e sgradite, ed avere una flessibilità nell' adozione di un quarzo piuttosto che un' altro. Il PC che state utilizzando ora viaggia ad una frequenza che sarebbe problematica da ottenere esternamente alla CPU, viene quindi anch' essa ottenuta internamente tramite un PLL.
Il PLL di questo micro ha comunque una limitazione che è dettata anche dalla

ragione per cui esiste: fornire un clock a 48MHz per l' interfaccia USB. Fortuna vuole che questo bel 48MHz lo possiamo anche utilizzare per dare il clock al micro. La limitazione sta nel fatto che questo signore, il PLL, accetta una sola frequenza di ingresso: 4MHz. Da questi 4 MHz ne tira fuori 96 che poi divisi per 2 danno un 48MHz pulito.
C' è però una cosa da sapere: il PLL lo si deve fare partire. Ci sono micro meno evoluti che non hanno bisogno di far partire da programma il PLL ma, di contro, non hanno neanche la possibilità di spegnerlo e di far viaggiare il micro ad una frequenza inferiore per esempio. Non sempre si vuole la velocità massima, a volte bisogna tenere bassi i consumi e, visto che più il micro va veloce e più consuma, avere una sezione di clock dove poterci cambiare la frequenza durante l' esecuzione del programma è una cosa non da poco. Nel nostro caso se vogliamo mettere il turbo al micro lo dobbiamo fare con una semplice istruzione, vedremo più avanti come.
Ricordiamoci solo che il **PIERIN PIC18** monta un quarzo da 12MHz e, alla partenza, funzionerà con un clock a 12MHz. Quindi non funziona alla velocità massima ma ben 4 volte più lento.

La base tempi

Abbiamo bisogno di una base tempi che generi una interrupt ogni millisecondo. Potremmo benissimo utilizzare un circuito esterno che generi un' onda quadra alla frequenza di 1KHz, magari partendo da un oscillatore con un cristallo in modo da avere precisione e che, collegato ad un piedino del micro, ci generi una interrupt. Di sicuro non ci sogneremmo di utilizzare un monostabile retriggerabile che dia in uscita un impulso da 1 ms. e che deve essere fatto partire ogni impulso, mi pare ovvio. Il circuito della base tempi deve esse un qualcosa che funziona autonomamente.
Utilizzando un microcontrollore sarebbe bello che tutto questo lavoro fosse fatto da un circuito interno senza aver bisogno di altro. E, guarda un po', manco a farlo a posta il micro ha a bordo questo circuito, o meglio, ha diversi timer ed alcuni di questi sono perfetti per lo scopo. Per questo tipo di lavoro si utilizza un timer particolare (neanche poi tanto), il timer con il registro di comparazione. E' preciso, funziona autonomamente, lo si inizializza una volta sola e poi lui continuerà a generare una interrupt ciclicamente.
Forse qualcuno ha sentito parlare di un altro tipo di timer: quello che lavora sull' overflow. E' un tipo di timer che si comporta come un monostabile e quindi bisogna ricaricarlo ogni volta che genera una interrupt. Va bene per fare ritardi singoli (one-shot) e volendo anche per fare una base tempi ma non è indicato. Il motivo sta nel fatto che, visto che deve essere ricaricato tutte le volte, le operazioni di ricarica (le istruzioni che il micro esegue per ricaricare il timer) introducono ritardi. Questi ritardi si potrebbero compensare opportunamente ma è un lavoro più complicato. Oddio, alcuni micro (sopratutto quelli di una volta) hanno solo questo tipo di timer perché è più semplice da realizzare nel silicio. In tal caso non si ha scelta, si fa quello

che si deve fare con quello che si ha a disposizione, ma **noi l' alternativa ce l' abbiamo e quindi la usiamo**.

Il timer 2

E' un timer con il registro di comparazione, perfetto per implementare una base tempi, vediamo come. Gli si scrive dentro il registro comparatore il numero di conteggi che deve fare per generare una interrupt. In pratica lui parte da zero e quando raggiunge il valore del registro comparatore si da da solo un reset, torna a zero e genera l' interrupt. E lo fa autonomamente. E' ovvio che ha a disposizione una serie di opzioni (presacaler e postscaler) per poter generare ritardi brevi o lunghi ma la cosa che mi preme sottolineare è proprio il fatto che funziona autonomamente, non ha bisogno della CPU per fare quello che deve fare ed è preciso.

Il programma

E' venuto il momento di analizzare il programma nel dettaglio. Riassumento avremo un programma composto da:

- Un main (ovviamente) dove si trovano:
  - Le istruzioni per accendere il PLL per fare girare il micro alla massima velocità
  - le inizializzazioni della porta dei LED e dei pulsanti
  - L' inizializzazione del timer 2 che farà da base tempi.
  - Le istruzioni per abilitare le interrupt
  - Il ciclo infinito di funzionamento che gestisce il lampeggio.
- Una funzione di servizio della interrupt che implementa il timer software
- Una variabile globale che noi utilizzeremo come timer software

Ovviamente anche la sezione di configurazione ma quella è già bella e pronta. Lascio al lettore (e la consiglio caldamente) l' analisi del file di configurazione per prendere dimestichezza con le opzioni del microcontrollore.

La funzione void main(void)

Per prima cosa troviamo la dichiarazione di una variabile che ci servirà più avanti. Il suo significato lo vedremo dopo. Sarebbe sto bello dichiararla più avanti ma il C vuole che le variabili siano dichiarate prima del corpo della funzione e quindi dobbiamo farlo qui, non abbiamo scelta.

```
char c;  // utizzata per la durata dei lampeggi
```

Subito dopo troviamo le istruzioni per far partire il PLL. Da notare che è stato inserito un ritardo "hardware". Lo scopo di tale ritardo (che è addirittura esagerato) è quello

di lasciare il tempo al PLL di stabilizzare l' oscillazione. Infatti (come scritto nel Data Sheet) il clock non sarà quello del PLL fino a quando questo non sarà stabile. Ma noi non abbiamo fretta ed aspettiamo volentieri che il PLL si stabilizzi con tutta calma e tranquillità.

```
// Fa partire il PLL.
// Anche se viene selezionato tramite i bit di configurazione
// il suo funzionamento non è automatico. Ha bisogno di un comando.
OSCTUNEbits.PLLEN = 1;
// Attende abbastanza tempo per far stabilizzare il PLL
Delay1KTCYx(100);
// Da ora in poi abbiamo il PLL funzionante ed il micro con il turbo.
```

Quindi, messo il turbo al micro, continuiamo inizializzando gli ingressi e le uscite.

```
// Inizializza la PORTD
// bit 4 input pulsante PL0
//  "  5 input pulsante PL1
//  "  6 output LED LED1
//  "  7 output LED LED2
TRISD = 0x3F;
// Mette a 0 tutte le uscite
LATD = 0;
```

E quindi l 'inizializzazione del timer 2

```
// De-inizializza il timer2. Non sarebbe necessario perché il micro esce
// allo stato di RESET ma è comunque buona pratica de-inizializzare sempre
// le periferiche per non tralasciare nessun bit.
timer2_deInit();
// Inizializza il timer 2 per interrupt ogni millisecondo.
// prescaler divide per 16
T2CONbits.T2CKPS = 2;
// Postscaler divide per 5
T2CONbits.T2OUTPS = 4;
// Imposta il valore comparatore a 150
PR2 = 150;
// Imposta l' interrupt del Timer 2 a priorita' bassa
IPR1bits.TMR2IP = 0;
// abilita interrupt del timer
PIE1bits.TMR2IE = 1;
```

Della de-inizializzazione del timer ne parleremo più avanti.
Preoccupiamoci invece di selezionare ed abilitare le interrupt. In questo caso

utilizziamo la possibilità di dare una priorità alle interrupt. Quella del timer ha priorità bassa. Perché bassa? Non c'è un motivo particolare, è solo un' occasione per far notare che il micro ha questa possibilità. Un giorno potrà diventare utile e quindi è buona cosa studiarla ora.

```
  // Ora che si sono inizializzate tutte le periferiche si possono abilitare
  // Oppurtunamente le interrupt
  // abilita le interrupt a bassa priorita'
  RCONbits.IPEN = 1;
  // abilta tutte le interrupt a priorità bassa
  INTCONbits.GIEL = 1;
  // Abilita tutte le interrupt in generale
  INTCONbits.GIEH = 1;
```

E' tutto a posto, il micro viaggia veloce, tutto è inizializzato e quindi non ci resta che far partire il timer. Questo continuerà a funzionare per gli affari suoi.

```
// Con le interrupt abilitate possiamo ora far partire il timer 2
  // Accende il timer
  T2CONbits.TMR2ON = 1;
```

Come ultima cosa prima di tuffarci nel ciclo infinito di funzionamento: inzializzare il timer software e la viariabile di conteggio.

```
  // Inizializza la variabile che funge da timer software
  timer_delay = 0;
  // Inizializza la variabile di conteggio
  c = 0;
```

Il ciclo infinito di funzionamento

Una cosa è necessario notare: il ciclo di funzionamento fa qualcosa solo quando il timer software è arivvato a zero, cioè quando è passato il tempo impostato. Se il timer software ha un valore diverso da 0 il programma salta di netto tutta la parte di gestione per ... fare niente. Si, perché in questo programmino non ha nient' altro di meglio da fare ma in programmi più complessi, se il timer software non è a zero, lui magari farebbe altre cose, ad esempio gestire la linea seriale, oppure implementare una qualsiasi altra cosa. E' per questo motivo che si usano i timer e non i ritardi fissi. Se il timer non ha raggiunto il valore voluto il microcontrollore è libero di fare altro.

```
  // -------------- Ciclo infinito di funzionamento -------------
  for(;;)
  {
```

```
// Se il timer software è arrivato a 0 esegue la gestione del lampeggio
if(!timer_delay)
{
  // Ricarica il timer software per ritardo di 200 ms.
  timer_delay = 200;

  // Guarda se i due pulsanti sono rilasciati. Nel caso lo siano
  // I due LED devono lampeggiare alternativamente con periodo
  // di 800 ms.
  if (PORTDbits.RD4 && PORTDbits.RD5)
  {
    // Si controlla il bit 2 perché cambia ogni 400 ms.
    if (c & 0x02)
    {
      LATDbits.LATD6 = 1;
      LATDbits.LATD7 = 0;
    }
    else
    {
      LATDbits.LATD6 = 0;
      LATDbits.LATD7 = 1;
    }
  }
  else

  // Uno o più pulsanti sono premuti. In questo caso deve lampeggiare
  // il LED che è posto sopra il pulsante con periodo 400 ms.
  {
    // Spegne i due LED
    LATDbits.LATD7 = 0;
    LATDbits.LATD6 = 0;

    // Guarda se è premuto il tasto PL0
    if (!PORTDbits.RD4)
    {
      if (c & 0x01)
      {
        LATDbits.LATD6 = 1;
      }
      else
      {
        LATDbits.LATD6 = 0;
      }
```

```
        } // if (!PORTDbits.RD4)

        // Guarda se è premuto il tasto PL1
        if (!PORTDbits.RD5)
        {
          if (c & 0x01)
          {
            LATDbits.LATD7 = 1;
          }
          else
          {
            LATDbits.LATD7 = 0;
          }
        } // if (!PORTDbits.RD5)
      }

      // Incrementa la variabile di conteggio
      // il suo valore deve andare da 0 a 3 quindi se vale 4 la si
      // riporta a 0
      c++;
      if (4 == c) c=0;
    } // if(!timer_delay)

  } // for(;;)
}
```

E qui vediamo a cosa serve la viabile **c**. In pratica si usa come una base tempi. Infatti il suo compito è contare da 0 a 3. Leggendo i suoi due bit più bassi controlliamo la frequenza dei lampeggi.

La variabile che funge da timer software

E' dichiarata come variabile globale

```
volatile unsigned short timer_delay;    // Timer software
```

- **volatile** perché il suo valore può cambiare da un momento all' altro. I compilatori cercano di generare codice veloce e compatto. L' attributo "volatile" obbliga il compilatore ad andare veramente a leggere il valore di una variabile quando questa viene utilizzata.
- **unsigned** perché non andrà mai negativa. Si comporta come un timer che si decrementa fino ad arrivare a zero per poi fermarsi. Inoltre il compilatore genera codice più compatto e veloce se si utilizzano variabili "unsigned"

- **short** (sottointeso int) per dichiararla come un intero a 16 bit. Ma allora perché non dichiararla semplicemente come "int"? Per questo compilatore gli "int" sono a 16 bit ma se stessimo lavorando con un micro a 32 bit molto probabilmente il compilatore userebbe una variabile a 32 bit. La cosa ci interessa perché è una buona abitudine scrivere senza dare niente per scontato. Quindi, anche se sappiamo che il compilatore utilizzerebbe comunque una variabile a 16 bit, noi lo dichiariamo apertamente perché siamo meticolosi, puntigliosi ed anche rompiscatole e tignosi.

La funzione di servizio della interrupt

```
//-----------------------------------------------------------------------------
// Funzione di servizio delle interrupt a BASSA priorità
//-----------------------------------------------------------------------------
#pragma interruptlow lowPriorityInterrupt
void lowPriorityInterrupt()
{
  // Verifica quale flag ha causato l' interrupt
  // Esegui la parte di codice di servizio dell' interrupt
  // Azzera il flag che ha causato l' interrupt
  // ...

  // Gestione dell' interrupt del timer 2
  if(PIR1bits.TMR2IF)
  {
    // gestione del timer software. Il timer software deve decrementarsi
    // fino ad arrivare a 0. Una volta arrivato a 0 resta fermo a 0.
    if (timer_delay) timer_delay--;

    // Resetta il flag che ha generato l' interrupt
    PIR1bits.TMR2IF = 0;
  }
}
```

E' molto semplice. Per prima cosa controlla che l' interrupt sia stata generata proprio dal timer 2 guardando il flag a posta il bit **TMR2IF** del registro **PIR1**, poi semplicemente se la variabile **timer_delay** è diversa (in pratica maggiore) di zero la decrementa, altrimenti se è a zero la lascia così com'è. Come ultima operazione resetta il flag dell' interrupt. Se non lo si fa la prossima interrupt non verrà riconosciuta.

Una curiosità: la funzione di de-inizializzazione del timer

C' è questa strana funzione che sembra non servire a niente.

```
//-----------------------------------------------------------------------------
// De-inizializza il timer 2 e lo porta nello stato in cui si trovava
// subito dopo il RESET
void timer2_deInit(void)
{
  T2CON = 0;            // Resetta il timer 2 control register
  TMR2 = 0;             // Azzera il contatore interno
  PR2 = 0;              // Azzera il registro comparatore
  PIE1bits.TMR2IE = 0;  // Disabilita l' interrupt
  IPR1bits.TMR2IP = 0;  // Resetta il bit di priorità dell' interrupt
  PIR1bits.TMR2IF = 0;  // Azzera il flag di interrupt
}
```

E' stata scritta più che altro per uno scopo didattico.
Sappiamo bene che il timer 2 dopo il reset è ancora da inzializzare, perché dunque chiamare una funzione di de-inizializzazione? Per essere sicuri che, il programma dovesse re-inizializzare il timer 2 per altri scopi, chiamando questa funzione saremmo sicuri di ritrovarcelo nello stato che avrebbe subito dopo il reset.
In effetti lavorando su un micro semplice una funzione del genere potrebbe anche non essere scritta ma quando si lavora con i bestioni (che di opzioni ne hanno una lista lunga come la fame) è praticamente obbligatorio scrivere la funzione di de-inizializzazione proprio per essere sicuri che qualche bit/opzione non si trovi in uno stato che magari creerebbe problemi o farebbe funzionare male la periferica. Per de-inizializzare il timer 2 bastano poche linee di programma ma invito il lettore ad andarsi a vedere cosa sono e quante opzioni hanno le periferiche di micro più grandi: hanno talmente tante opzioni che ... mamma mia!
Quindi prendere la buona abitudine di scrivere queste funzioni è una buona cosa, utile per scrivere programmi affidabili ed indispensabile quando si passerà a lavorare con micro più potenti e complessi.

Il sorgente completo

```
// File di definizione dei registri del micro.
#include "p18f47j53.h"

// File di configurazione dei fuses
#include "configurazione.h"

// Mappatura delle interrupt
```

```
#include "mappa_int.h"

// Header del main
#include "main.h"


//-----------------------------------------------------------------------------
// Variabili globali
//-----------------------------------------------------------------------------
#pragma udata
volatile unsigned short timer_delay;    // Timer software


//-----------------------------------------------------------------------------
// Funzione di servizio delle interrupt ad ALTA priorità
//-----------------------------------------------------------------------------
#pragma code
#pragma interrupt highPriorityInterrupt
void highPriorityInterrupt()
{
  // Verifica quale flag ha causato l' interrupt
  // Esegui la parte di codice di servizio dell' interrupt
  // Azzera il flag che ha causato l' interrupt
  // ...
}

//-----------------------------------------------------------------------------
// Funzione di servizio delle interrupt a BASSA priorità
//-----------------------------------------------------------------------------
#pragma interruptlow lowPriorityInterrupt
void lowPriorityInterrupt()
{
  // Verifica quale flag ha causato l' interrupt
  // Esegui la parte di codice di servizio dell' interrupt
  // Azzera il flag che ha causato l' interrupt
  // ...

  // Gestione dell' interrupt del timer 2
  if(PIR1bits.TMR2IF)
  {
    // gestione del timer software. Il timer software deve decrementarsi
    // fino ad arrivare a 0. Una volta arrivato a 0 resta fermo a 0.
    if (timer_delay) timer_delay--;
```

```
    // Resetta il flag che ha generato l' interrupt
    PIR1bits.TMR2IF = 0;
  }
}

//-----------------------------------------------------------------------------
// Prototipi delle funzioni
//-----------------------------------------------------------------------------
#pragma code
void timer2_deInit(void);

//-----------------------------------------------------------------------------
// Funzioni
//-----------------------------------------------------------------------------
#pragma code

//-----------------------------------------------------------------------------
// De-inizializza il timer 2 e lo porta nello stato in cui si trovava
// subito dopo il RESET
void timer2_deInit(void)
{
  T2CON = 0;            // Resetta il timer 2 control register
  TMR2 = 0;             // Azzera il contatore interno
  PR2 = 0;              // Azzera il registro comparatore
  PIE1bits.TMR2IE = 0;  // Disabilita l' interrupt
  IPR1bits.TMR2IP = 0;  // Resetta il bit di priorità dell' interrupt
  PIR1bits.TMR2IF = 0;  // Azzera il flag di interrupt
}

//-----------------------------------------------------------------------------
// MAIN
void main(void)
{
  char c;  // utizzata per la durata dei lampeggi

  // Fa partire il PLL.
  // Anche se viene selezionato tramite i bit di configurazione
  // il suo funzionamento non è automatico. Ha bisogno di un comando.
  OSCTUNEbits.PLLEN = 1;
  // Attende abbastanza tempo per far stabilizzare il PLL
  Delay1KTCYx(100);
  // Da ora in poi abbiamo il PLL funzionante ed il micro con il turbo.
```

```
// -------- Inizializzazione delle periferiche --------

// Inizializza la PORTD
// bit 4 input pulsante PL0
//  "  5 input pulsante PL1
//  "  6 output LED LED1
//  "  7 output LED LED2
TRISD = 0x3F;
// Mette a 0 tutte le uscite
LATD = 0;


// De-inizializza il timer2. Non sarebbe necessario perché il micro esce
// allo stato di RESET ma è comunque buona pratica de-inizializzare sempre
// le periferiche per non tralasciare nessun bit.
timer2_deInit();
// Inizializza il timer 2 per interrupt ogni millisecondo.
// prescaler divide per 16
T2CONbits.T2CKPS = 2;
// Postscaler divide per 5
T2CONbits.T2OUTPS = 4;
// Imposta il valore comparatore a 150
PR2 = 150;
// Imposta l' interrupt del Timer 2 a priorita' bassa
IPR1bits.TMR2IP = 0;
// abilita interrupt del timer
PIE1bits.TMR2IE = 1;

// -------- Selezione ed abilitazione delle interrupt --------
// Ora che si sono inizializzate tutte le periferiche si possono abilitare
// Oppurtunamente le interrupt
// abilita le interrupt a bassa priorita'
RCONbits.IPEN = 1;
// abilta tutte le interrupt a priorità bassa
INTCONbits.GIEL = 1;
// Abilita tutte le interrupt in generale
INTCONbits.GIEH = 1;

// -------- Attivazione delle periferiche --------
// Con le interrupt abilitate possiamo ora far partire il timer 2
// Accende il timer
T2CONbits.TMR2ON = 1;
```

```
// Inizializza la variabile che funge da timer software
timer_delay = 0;
// Inizializza la variabile di conteggio
c = 0;

// -------------- Ciclo infinito di funzionamento ------------
for(;;)
{

  // Se il timer software è arrivato a 0 esegue la gestione del lampeggio
  if(!timer_delay)
  {
    // Ricarica il timer software per ritardo di 200 ms.
    timer_delay = 200;

    // Guarda se i due pulsanti sono rilasciati. Nel caso lo siano
    // I due LED devono lampeggiare alternativamente con periodo
    // di 800 ms.
    if (PORTDbits.RD4 && PORTDbits.RD5)
    {
      // Si controlla il bit 2 perché cambia ogni 400 ms.
      if (c & 0x02)
      {
        LATDbits.LATD6 = 1;
        LATDbits.LATD7 = 0;
      }
      else
      {
        LATDbits.LATD6 = 0;
        LATDbits.LATD7 = 1;
      }
    }
    else

    // Uno o più pulsanti sono premuti. In questo caso deve lampeggiare
    // il LED che è posto sopra il pulsante con periodo 400 ms.
    {
      // Spegne i due LED
      LATDbits.LATD7 = 0;
      LATDbits.LATD6 = 0;

      // Guarda se è premuto il tasto PL0
```

```
        if (!PORTDbits.RD4)
        {
          if (c & 0x01)
          {
            LATDbits.LATD6 = 1;
          }
          else
          {
            LATDbits.LATD6 = 0;
          }
        } // if (!PORTDbits.RD4)

        // Guarda se è premuto il tasto PL1
        if (!PORTDbits.RD5)
        {
          if (c & 0x01)
          {
            LATDbits.LATD7 = 1;
          }
          else
          {
            LATDbits.LATD7 = 0;
          }
        } // if (!PORTDbits.RD5)
      }

      // Incrementa la variabile di conteggio
      // il suo valore deve andare da 0 a 3 quindi se vale 4 la si
      // riporta a 0
      c++;
      if (4 == c) c=0;
    } // if(!timer_delay)

  } // for(;;)
}
```

Una nota di colore. La quintultima linea è scritta in modo, diciamo, particolare. Perché invece di scrivere

```
if (c == 4)
```

è scritto questo?

```
if (4 == c)
```

Perché se per un errore di battitura scrivessi un solo "=" ,nel primo caso il programma non funzionerebbe ma il compilatore non darebbe errore. Il compilatore assegnerebbe 4 alla variabile c ed il risultato dell' assegnazione sarebbe 4. In buona sostanza la condizione sarebbe sempre verificata poiché il risultato dell' assegnazione è diverso da 0.
Nel secondo caso il compilatore darebbe errore perché non si può assegnare il valore contenuto in una variabile ad una costante numerica.
Sapendo che **uno dei trabocchetti più frequenti un cui ci si imbatte scrivendo sorgente in C è proprio quello di utilizzare un' operazione di assegnazione quando si vorrebbe fare un confronto**, questo semplice sistema evita di cadere in errore.
Anche questa sarebbe una buona abitudine da adottare.

Conclusioni

E così abbiamo anche fatto le pulci al programma di demo.
E' stata un' occasione buona per imparare qualcosa di utile (almeno questa era l' intenzione) sfruttando un progetto semplice. Spero che possa essere utile a qualcuno, sopratutto ai neofiti, e che sia un modo per poter prendere da subito buone abitudini.
Non mi resta quindi, come al solito, di augurarvi una **BUONA SPERIMENTAZIONE!**

P.S.: Mi scuso per gli errore e le imprecisioni che ci potrebbero essere in questo articolo. Se ne trovate vi prego di dirmelo e li correggerò.

Estratto da "http://www.electroyou.it/mediawiki/index.php?title=UsersPages:Tardofreak:pierin-pic18-il-programma-demo-timer-e-super-velocit-per-il-micro"