SVILUPPARE CON I PIC: Primo programma con MPLAB X e XC8

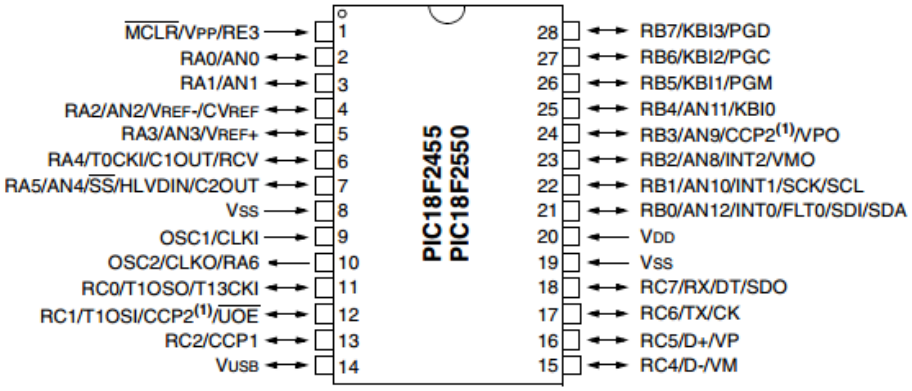
Vedremo come scrivere un primo programma per il nostro microcontrollore, ma, prima di mettere mano al codice, abbiamo bisogno di conoscere almeno in minima parte l'hardware del PIC che andremo ad utilizzare e il circuito minimale che consente di farlo funzionare.
Un microcontrollore è un circuito integrato simile ad un microprocessore, dunque possiede un'unità di controllo, un'unità aritmetico-logica, dei registri, dei bus interni, ecc., ma non solo. Al suo interno sono presenti anche tutta una serie di periferiche ed interfacce che gli permettono di comunicare con il mondo esterno. Potremo avere ad esempio un convertitore analogico-digitale (ADC), un modulo pulse-width modulation (PWM), dei timer, un modulo USB e così via, cose che rendono il microcontrollore un dispositivo completo ed autosufficiente per la maggior parte delle applicazioni. I microcontrollori sono utilizzati all'interno dei cosiddetti sistemi embedded, ovvero quei sistemi pensati per assolvere ad una specifica funzione (come il controller di una lavatrice), che si differenziano dai sistemi di tipo general purpose, come lo è un personal computer, adibiti invece a compiti che possono essere i più svariati possibile. Dopotutto, nelle applicazioni embedded tutta la potenza di calcolo delle moderne CPU è inutile, dove invece la dissipazione di potenza e l'integrazione della circuiteria di contorno sono dei requisiti fondamentali.
Come ho già accennato nell'articolo precedente, useremo un microcontrollore a 8 bit della famiglia PIC18F, in particolare il PIC18F2550. Essendo queste delle guide introduttive avrei potuto impiegare benissimo un PIC di fascia media o bassa (PIC10/12/16). I motivi che mi hanno portato ad usare questo modello sono principalmente due: il primo, più banale, è che esso è il PIC che conosco meglio ed è tra i più usati in generale, in quanto contiene una grande varietà di periferiche. Nella fase di sviluppo di un'applicazione infatti è consigliabile utilizzare il dispositivo più grande che si possa avere a disposizione, sia in termini di quantità memoria che di periferiche integrate, per non rischiare di "rimanere a piedi" nel bel mezzo del progetto. Una volta che il progetto sarà stato completato, oppure in una sua fase avanzata, si potrà scegliere il microcontrollore che più si adatta allo scopo (periferiche e memoria non sfruttate sono uno spreco in termini di costo). Il secondo motivo, più importante, è la presenza del modulo USB integrato. Tale presenza è fondamentale, in quanto tra qualche lezione andremo ad implementare un bootloader all'interno del microcontrollore, che ci consentirà di riprogrammarlo al volo, senza usare alcun tipo di programmatore. Ero intenzionato a introdurre già in questa lezione il bootloader, tuttavia esso nasconde alcuni dettagli sull'hardware (come ad esempio i configuration bits), per cui non mi è sembrato didattico parlarne subito. Nonostante ciò il circuito che presenteremo sarà già pronto per il funzionamento con bootloader.

L'hardware del microcontrollore

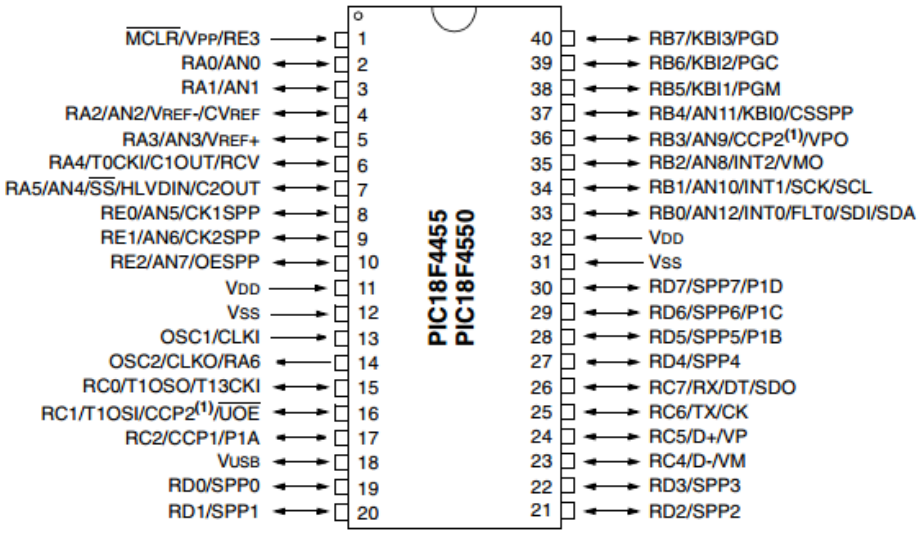
Passiamo a questo punto a descrivere un po' meglio le caratteristiche del PIC18F4550. La prima cosa che dobbiamo fare è scaricare il datasheet del microcontrollore all'indirizzo http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010300. Durante questi articoli farò spesso riferimento alle pagine del datasheet, che quindi diventa il nostro punto di rifermento ufficiale per l'hardware. Per chi è alle prime armi consultare un datasheet di circa 500 pagine (e in inglese) potrebbe risultare scoraggiante, per cui consiglio di leggere prima il Capitolo II del già citato C18 Step by Step di Mauro Laurenti, che affronta l'argomento in maniera molto più amichevole. In rete inoltre è disponibile un tentativo di traduzione in italiano del datasheet di un PIC simile al nostro (il PIC18F2321, che manca del modulo USB): http://www.microcontroller.it/Tutorials/PICbyDS/000_indice.htm. Può essere consultato da chi ha difficoltà con la lingua anglosassone.

Allora, il PIC18F2550 è disponibile sia in package PDIP da 28 pin che in TQFP e QFN da 28 pin. Ovviamente consiglio di prendere la versione PDIP, che possiamo montare anche su breadboard. Il nostro PIC può lavorare ad una frequenza operativa massima di 48 MHz (12 MIPS), ha 32 KB di memoria flash, 2 KB di RAM, una EEPROM interna di 256 byte, 2 comparatori, 4 timer, 1 modulo ADC da 10 bit e 10 canali, 1 modulo MSSP, 1 modulo USB e altro ancora. Come notiamo dal pinout del PIC, mostrato in figura 1, le funzionalità del microcontrollore sono molto maggiori del numero di pin di I/O (Input/Output) messi a disposizione, per cui quasi tutti i pin hanno funzionalità multiple. Ovviamente tutte queste funzioni non possono essere usate contemporaneamente, ma in un determinato istante di funzionamento del microcontrollore ogni pin potrà avere una ed una sola funzione.

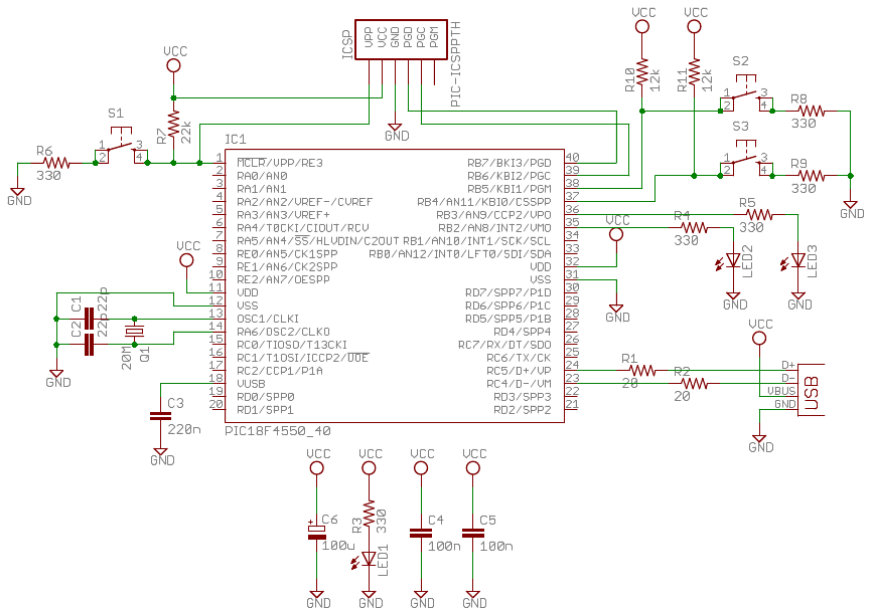
28-Pin PDIP, SOIC

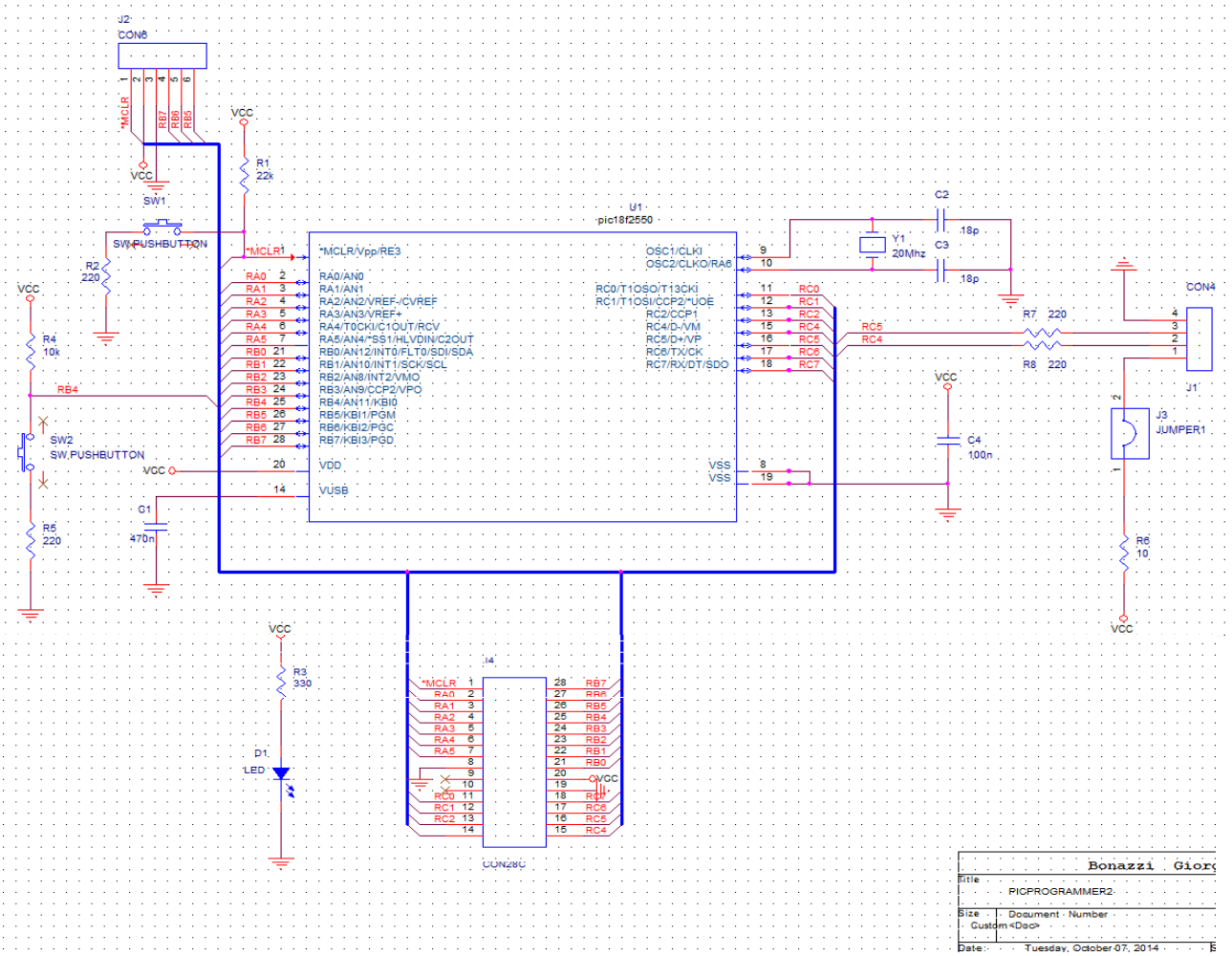


40-Pin PDIP



Dunque per interfacciarsi con il mondo esterno il PIC usa i pin di I/O, in gergo le “porte” del microcontrollore. Tali porte vengono descritte nel capitolo 10 del datasheet. Ogni porta (identificata da una lettera: A, B, C, D, E) è corredata da una circuiteria che permette di configurarla o come input o come output. Ovviamente una volta che la porta è stata configurata come input o come output, essa rimane tale per tutta la durata del programma. All’accensione tutte le porte sono impostate per default come input; ciò per evitare conflitti con eventuale hardware esterno che potrebbe imporre una determinata tensione sul pin. Vedremo tra un po’ come configurare le porte del microcontrollore nel codice.
Il circuito di prova che useremo è mostrato in figura 2. Effettivamente non si tratta di un circuito minimale, questo perché è già pronto per l’uso con bootloader, che richiederà la porta USB per la programmazione del PIC. Notiamo che l’alimentazione (VCC) è prelevata dal connettore USB (5 V) e va collegata ai pin del microcontrollore indicati con VDD. I pin del PIC indicati con VSS vanno invece messi a massa e tra VCC e massa vanno collegati due condensatori da 100 nF, il cui scopo è compensare eventuali fluttuazioni dell’alimentazione e che sono da posizionare il più vicino possibile ai pin VDD e VSS del PIC. Consiglio poi di mettere un condensatore elettrolitico a monte dell’alimentazione prelevata dall’USB, anche se questa è stabilizzata. Abbiamo anche un led (LED1, con relativa resistenza R3 in serie) il cui scopo è indicare la presenza dell’alimentazione. È possibile alimentare il circuito anche con un alimentatore esterno se lo si desidera, basta scollegare l’alimentazione della porta USB, avere l’accortezza di collegare insieme le masse dell’alimentatore e dell’USB ed assicurarsi che la tensione erogata dall’alimentatore sia di 5 V stabilizzati.

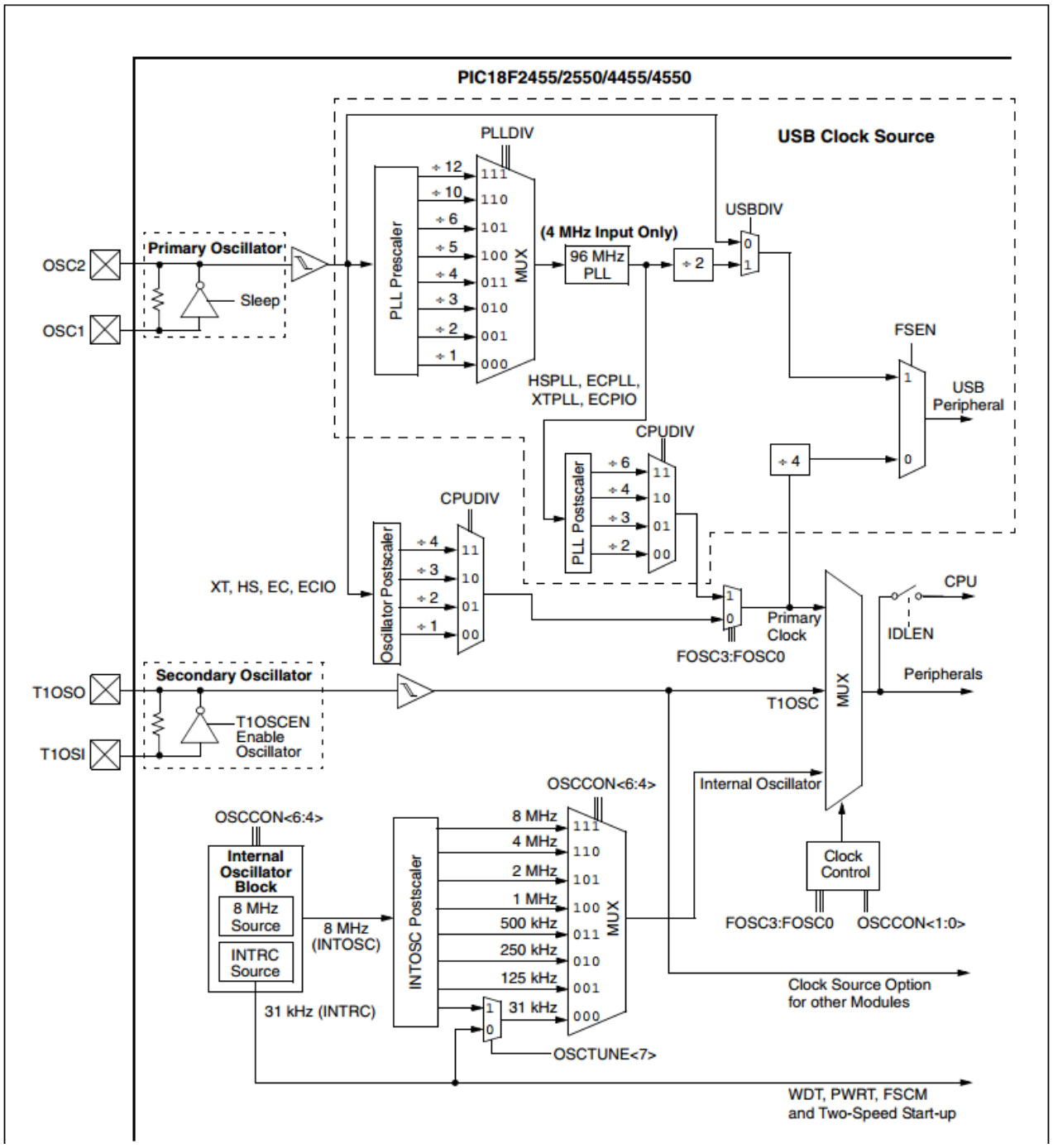




Notiamo sul pin 1 la presenza di un resistore di pull-up, che porta tale pin al valore logico alto, ed un tasto, che permette di portarlo al valore logico basso. Il pin 1 è un pin attivo basso noto come Master Clear, e serve ad effettuare il Reset del dispositivo, ovvero portare tutti i registri al loro valore di default. In pratica il reset avviene quando il valore logico di questo pin passa da basso ad alto, ed è ciò che si ha all'accensione. Infatti prima di collegare l'alimentazione il valore logico presente all'ingresso del pin MCLR è 0, dopodiché si porta a 1 e avviene il reset del dispositivo. Si parla di Power-on Reset. Ora premendo il tasto e rilasciandolo si ha lo stesso effetto del Power-on Reset.
La resistenza R7 di pull-up di solito ha un valore intorno ai 10 kΩ, ma ho preferito metterla più grande per un motivo che spiegherò a breve. La resistenza R2 poi è solo di protezione e può essere omessa nel caso di montaggio su breadboard.
Il nostro PIC non funziona se non vi è un clock in grado di fornire le giuste temporizzazioni. Sono possibili diverse configurazioni: la prima è quella di usare un clock generato esternamente, la seconda quello di usare il generatore di clock interno al PIC, la terza è quella di fare uso di un quarzo per innescare le oscillazioni. Noi utilizziamo la terza soluzione. In particolare nel circuito vediamo la presenza di un quarzo da 20 MHz collegato tra i pin 9 (OSC1) e 10 (OSC2); inoltre ognuno di questi pin va collegato a massa con dei condensatori il cui valore dipende dalla frequenza di oscillazione del quarzo. Il datasheet suggerisce un valore di 15 pF per una frequenza da 20 MHz, ma io avevo solo condensatori da 18 pF e quelli ho messo. Il motivo per cui non abbiamo usato un quarzo di frequenza inferiore (spesso si trovano progetti con quarzi da 4 o 8 MHz) è che noi abbiamo la necessità di generare internamente i 48 MHz necessari al funzionamento dell'USB in modalità full-speed. In figura 3 è mostrata una parte della sezione di generazione del clock di un PIC18F2550. Nel nostro caso i 20 Mhz del quarzo vengono prima divisi per 5 da un prescaler,

ottenendo un clock di 4 MHz, dopodiché esso è elevato a 96 MHz da un PLL ed infine diviso per 2. È possibile usare frequenze di funzionamento differenti per il core del microcontrollore e per il modulo USB, tuttavia abbiamo preferito usare la stessa frequenza per entrambi. Questa frequenza all'interno del datasheet è nota come FOSC, quindi nel nostro caso FOSC = 48 MHz. In realtà il PIC non funziona realmente a 48 MHz, in quanto il clock è caratterizzato da quattro fasi di frequenza pari a un quarto di quella originaria, e su ognuna di queste fasi viene eseguito il ciclo istruzioni del microcontrollore. Tale frequenza è nota come FCYC, per cui nel nostro caso FCYC = FOSC/4 = 12 MHz. Dunque per completare un ciclo istruzione sono necessari 4 cicli del clock principale.

FIGURE 2-1: PIC18F2455/2550/4455/4550 CLOCK DIAGRAM



Un altro componente fondamentale è il condensatore C1 da 470 nF (220 nF valore consigliato nel datasheet) posto sul pin 14 (VUSB), che serve per il regolatore di tensione interno del modulo USB.

Notiamo oltretutto che abbiamo collegato le linee dati D+ e D- dell'USB rispettivamente ai pin 16 e 15 del microcontrollore mediante delle resistenze da 220 ohm, che sono opzionali. Passiamo adesso alla descrizione cosiddetta Human Interface, ovvero l'interfaccia utente che consente l'interazione con il dispositivo. Al momento ho previsto un pulsante, ma negli articoli a seguire la espanderemo aggiungendo altre interfacce, come ad esempio un display LCD. Il pulsante invece è stato collegato sul pin 24 (RB4). In particolare il tasto su RB4 è di fondamentale importanza per l'uso con bootloader. La configurazione dei tasti è la stessa vista nel caso del reset, con una resistenza di pull-up intorno ai 10 kΩ e il pulsante collegato verso massa con una resistenza di protezione opzionale. In questa configurazione la condizione di tasto chiuso viene vista come uno 0 logico, mentre la condizione di tasto aperto come 1 logico.
L'ultimo elemento da analizzare nel nostro circuito è il connettore per il programmatore (header ICSP). In generale la programmazione dei PIC avviene in due modi: programmazione ad "alta" tensione (HVP), che consiste in pratica nell'applicare un segnale di circa 13 V al pin 1 del microcontrollore (la tensione nominale di funzionamento è di 5 V, per questo è alta), e programmazione a bassa tensione (LVP). Il connettore è standard ed è detto ICSP, che sta per In Circuit Serial Programming, ovvero programmazione seriale in circuit, cioè a circuito montato. Si tratta di una connessione prevista da tutti i microcontrollori Microchip e permette di effettuare la programmazione del dispositivo senza rimuoverlo dal circuito in cui si trova. Tale connessione è utile se si possiede un programmatore PicKit2 o 3, che usano appunto questo tipo di connessione. Il connettore prevede sul pin 1 la tensione di programmazione, detta VPP, di circa 13 V, che va collegata al pin 1 del microcontrollore. I pin 2 e 3 sono rispettivamente l'alimentazione (5 V) e la massa, che ovviamente vanno collegati ad alimentazione e massa del circuito, mentre i pin 4 e 5 sono la linea dati seriale e il clock, che vanno collegati rispettivamente ai pin 28 (PGD) e 27 (PGC) del microcontrollore. Nella nostra implementazione dell'ICSP viene utilizzata la programmazione HVP (quella che usano anche i PicKit). Se si vuole usare la LVP è necessario impiegare un pin in più, il pin 6 sull'header, che a noi è scollegato e che andrebbe collegato al pin 26 (PGM). Per maggiori approfondimenti sull'ICSP consiglio di consultare la seguente pagina web: http://www.microcontroller.it/Tutorials/PIC/icsp/icsp_icd.htm.
Veniamo ora al motivo per cui ho preferito aumentare il valore del resistore di pull-up sul reset. Quando si va a collegare il programmatore, questo andrà ad impostare in fase di programmazione una tensione di circa 13 V sul pin MCLR, portando ad un assorbimento di corrente da parte della circuiteria di alimentazione. Infatti il pin 1 del PIC si troverà ad una tensione maggiore rispetto a VCC, portando a circolare in questo nodo una corrente data dalla tensione sul pin 1 meno quella su VCC diviso la resistenza di pull-up. Aumentando la resistenza di pull-up riduciamo la corrente immessa nella porta USB al minimo, però non possiamo aumentarla a dismisura altrimenti la porta MCLR non riconoscerà più il valore logico alto. In ogni caso non si hanno problemi, in quanto la corrente assorbita dal circuito è maggiore di quella immessa nell'alimentazione (che si sottrae a quella assorbita), quindi non vi sarà corrente che andrà a fluire all'interno della porta USB. Al limite possiamo inserire un diodo contropolarizzato e non se ne parla più.

Cosa serve

Software

Il software richiesto, tutto gratuito anche se non libero, è disponibile sul sito Microchip. In particolare:

- MPLAB® XC8 Compiler (per Linux, Windows oppure Mac)
- MPLAB® X IDE (per Linux, Windows oppure Mac)

Hardware

Un *In-Circuit Debugger (ICD)*, cioè l'hardware da utilizzare per collegare il PC al PIC18; vanno bene, in alternativa:

- PICkit 3 In-Circuit Debugger - semplice e di costo relativamente basso
- PICkit 2 Development Programmer/Debugger (se lo avete già, non penso sia ancora in commercio)
- MPLAB ICD 3 In-Circuit Debugger - Permette di usare anche i breakpoint software

Serve inoltre:

- Una breadboard e relativi accessori
- (solo per alcune versioni di PIC18) la *debug header* specifica per il microcontrollore in uso
- (opzionale) uno dei tanti starter kit adatti al PIC18 che volete utilizzare

Documentazione

Per approfondire e integrare è necessario far riferimento, per esempio, ai seguenti documenti, disponibili soprattutto sul sito Microchip:

- MPLAB® XC8 C Compiler User's Guide
- MPLAB® XC8 Getting Started Guide
- MPLAB® X IDE User's Guide
- PIC18 Peripheral Library Help Document
- I datasheet del PIC18 che state utilizzando
- Altre note applicative specifiche presenti sul sito Microchip
- La grande mole di documentazione presente in rete
- Utile un qualunque manuale C, meglio se orientato all'hardware e/o ai sistemi embedded

Utile, anche se a volte un po' confuso, il forum Microchip

Gli starter kit

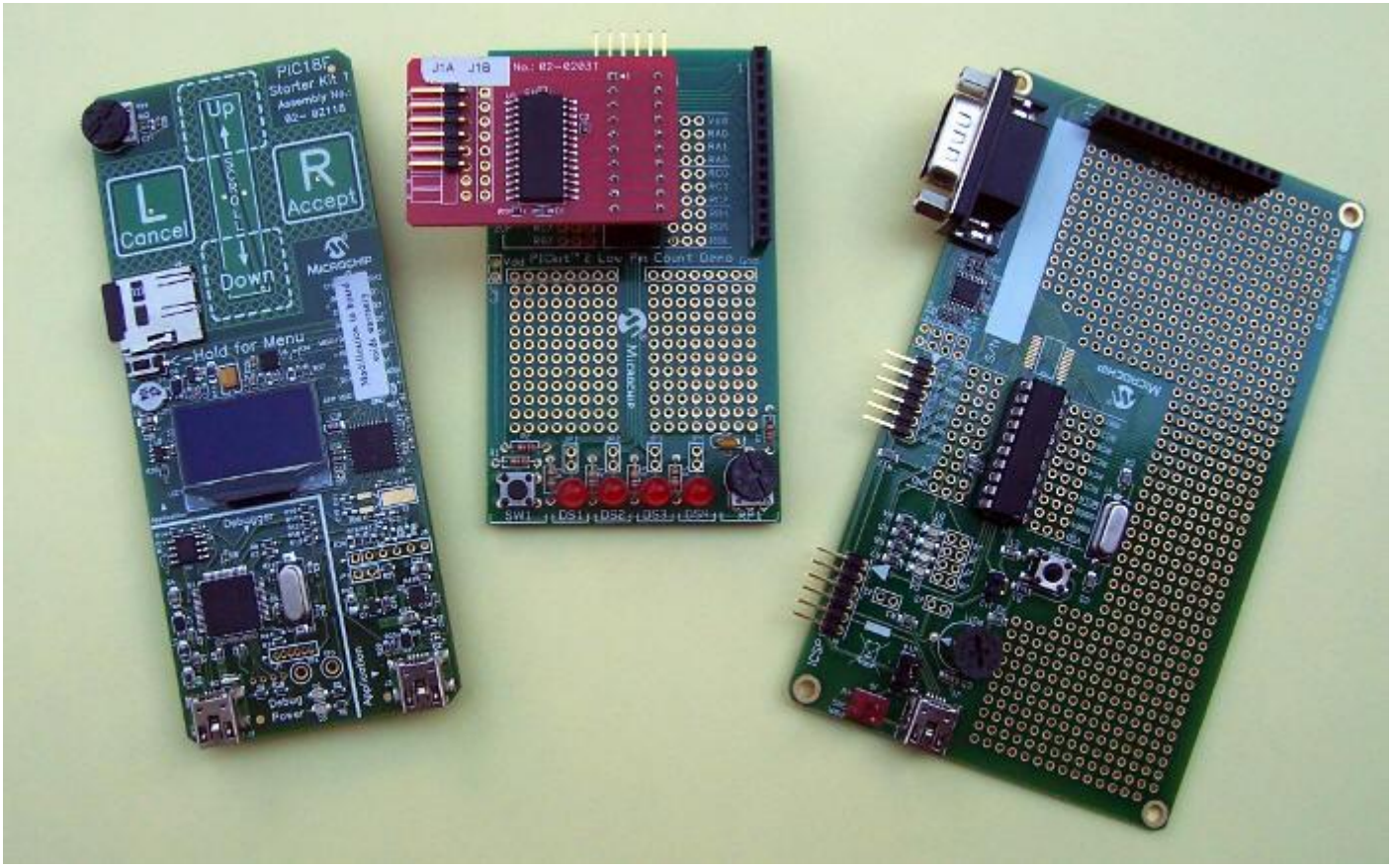
L'acquisto di uno starter kit è importante, ma assolutamente non obbligatorio, per una serie di motivi:

- rende disponibile un hardware già pronto all'uso. Se si tratta di pochi LED, ovviamente non è un grande vantaggio; diverso il caso in cui vi servono connettori USB, quarzi, display grafici e altre periferiche di reperimento non immediato
- permette, come opzione di avere un ICD a prezzo scontato. In genere oggi viene fornito il PICkit 3
- contiene, se necessario, la debug header, decisamente cara se acquistata separatamente
- (contiene il CD con MPLAB ed il software demo. Direi inutile, meglio scaricare l'ultima versione direttamente dal sito Microchip)

Personalmente ho utilizzato per i PIC18 diversi starter kit. Da sinistra:

- *MPLAB® Starter Kit for PIC18F MCUs*, un sacco di periferiche, ma un poco rigido perché usa un PIC18 SMD saldato. Non richiede un ICD separato
- *Low Pin Count Demo Board*, nato per il PIC16 ma adattabile anche ai PIC18 in contenitore a 20 pin. Se dovete acquistarlo, non ve lo consiglio per cominciare

- *Low Pin Count USB Development Kit*, utile soprattutto per sperimentare l'USB con il PIC18F14K50. Penso sia un buon compromesso tra funzionalità e costo



L'ambiente di sviluppo MPLAB X

Era il 2010 quando, a seguito di un contatto con **Microchip**, mi fu anticipato che nel giro di pochi mesi la casa di Chandler avrebbe rilasciato le prime versioni di **MPLAB** con caratteristiche innovative, tra le quali (forse la più ecclatante) il fatto di essere stato scritto in Java per renderlo disponibile per diversi sistemi operativo, **Windows**, **Linux** e **Mac**. L'idea di permettere agli *amici del pinguino* e della *mela morsicata* di poter utilizzare tutti gli strumenti Microchip per lo sviluppo dei progetti con i PIC avrebbe permesso di allargare notevolmente la platea di aficionados. Non nascondo il mio interesse verso Linux ma la mancanza di software specifici per la programmazione embedded (come i compilatori ed il supporto dei programmer/debugger) mi aveva frenato dall'utilizzo. Ora le cose stanno diversamente e quindi il passagio a Linux potrà essere meno doloroso e nel contempo compatibile con lo sviluppo embedded. Il nome di questo ambiente? **MPLAB X IDE**.

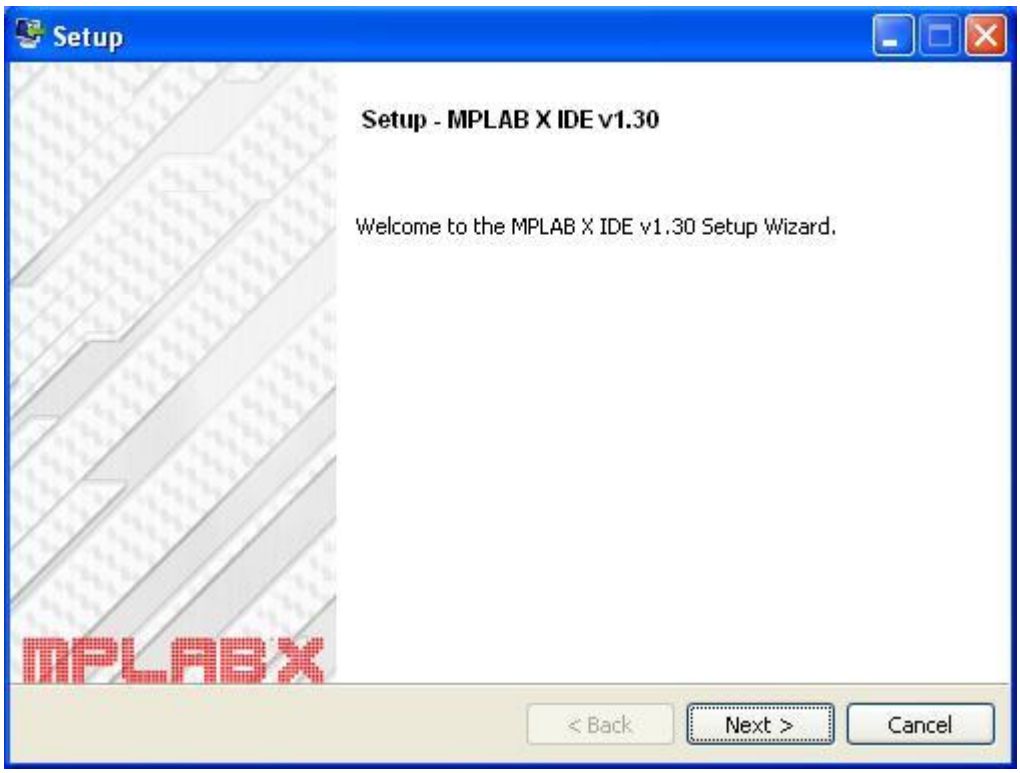
L'ultima versione

È sufficiente avviare il browser e scrivere http://www.mplab.com/mplab [1] per essere immediatamente indirizzati alla pagina di MPLAB X. Il link per scaricare la versione per il proprio sistema operativo. Oltre all'IDE è possibile anche scaricare le versioni X dei compilatori C per PIC16F, PIC18F, PIC24F, PIC32. Per i PIC18F è sempre possibile scaricare il C18.

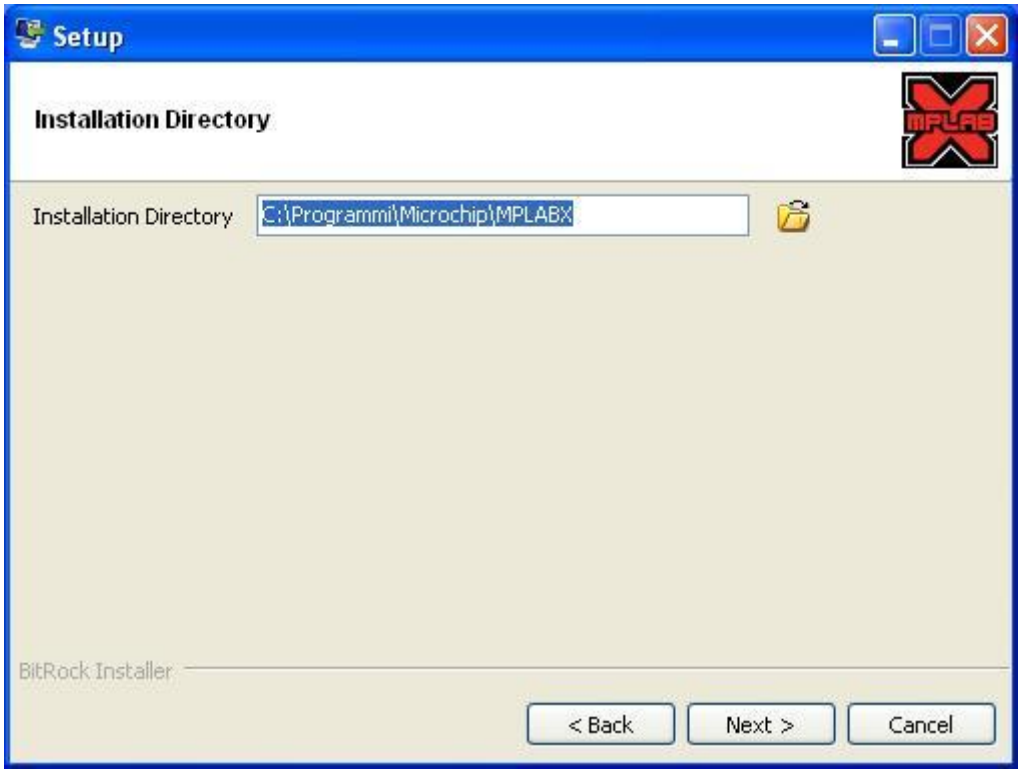
Una visita periodica a questa pagine permetterà di mantenere aggiornato MPLAB X. Questo aspetto non è un semplice sfizio: quando Microchip rende disponibili nuove funzionalità, oppure corregge bug individuati nelle versioni precedenti o semplicemente vengono rilasciati nuovi microcontrollori, allora è facile trovare una nuova versione dell'ambiente di sviluppo e può essere conveniente un aggiornamento.

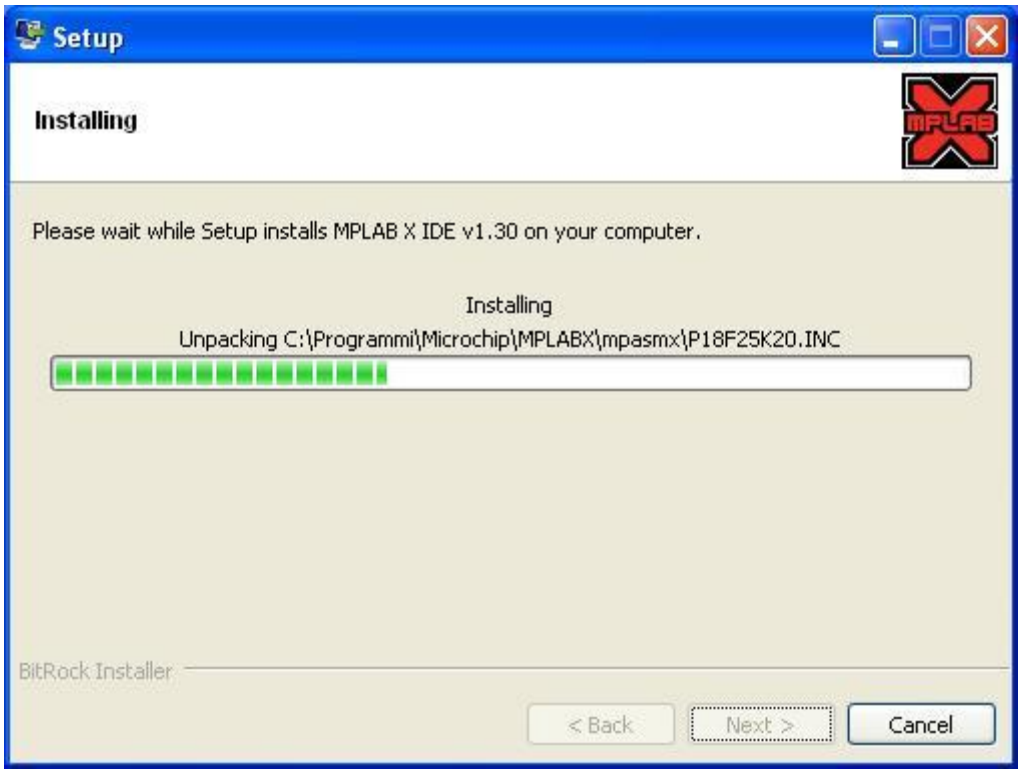
L'installazione.

La procedura di installazione su Windows è abbastanza immediata. Una volta lanciata la procedura, l'installazione si compone delle solite richieste, come l'accettazione della licenza e la scelta della cartella di destinazione.

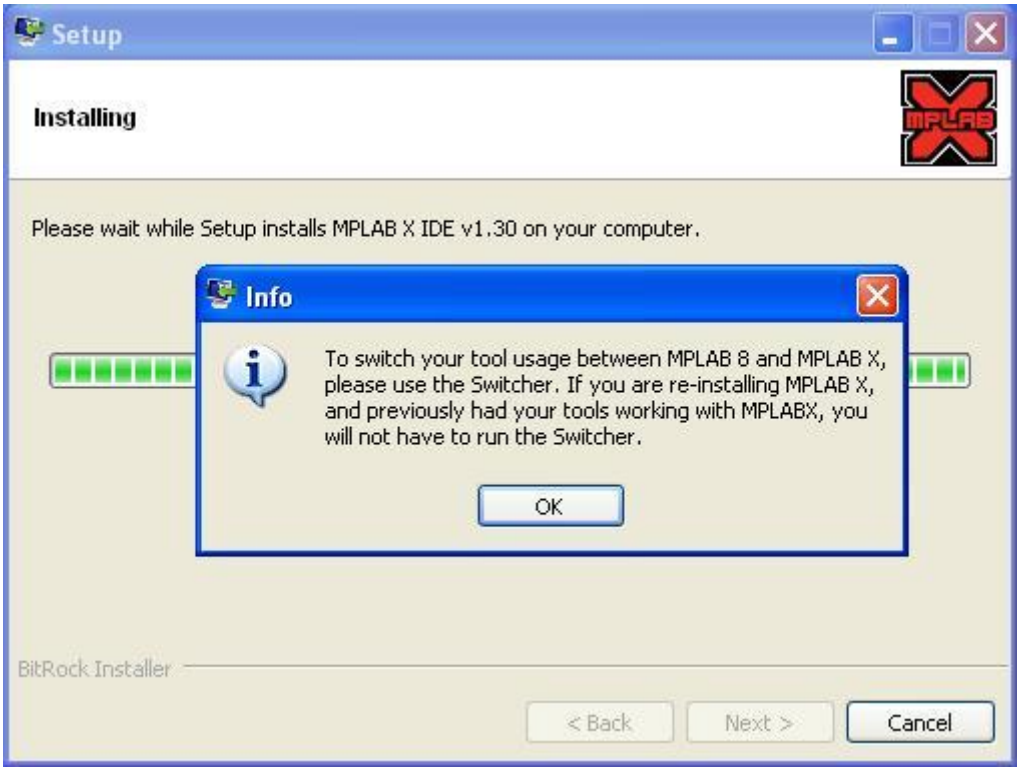

Setup
Setup - MPLAB X IDE v1.30
Welcome to the MPLAB X IDE v1.30 Setup Wizard.
MPLABX
< Back
Next >
Cancel


Setup
License Agreement
Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.
MICROCHIP IS WILLING TO LICENSE THE ACCOMPANYING SOFTWARE AND DOCUMENTATION TO YOU ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE FOLLOWING TERMS. TO ACCEPT THE TERMS OF THIS LICENSE, CLICK "I ACCEPT" AND PROCEED WITH THE DOWNLOAD OR INSTALL. IF YOU DO NOT ACCEPT THESE LICENSE TERMS, CLICK "I DO NOT ACCEPT," AND DO NOT DOWNLOAD OR INSTALL THIS SOFTWARE.
NON-EXCLUSIVE SOFTWARE LICENSE AGREEMENT FOR MPLAB(r) IDE SOFTWARE
Do you accept this license?
I accept the agreement
I do not accept the agreement
BitRock Installer
< Back
Next >
Cancel

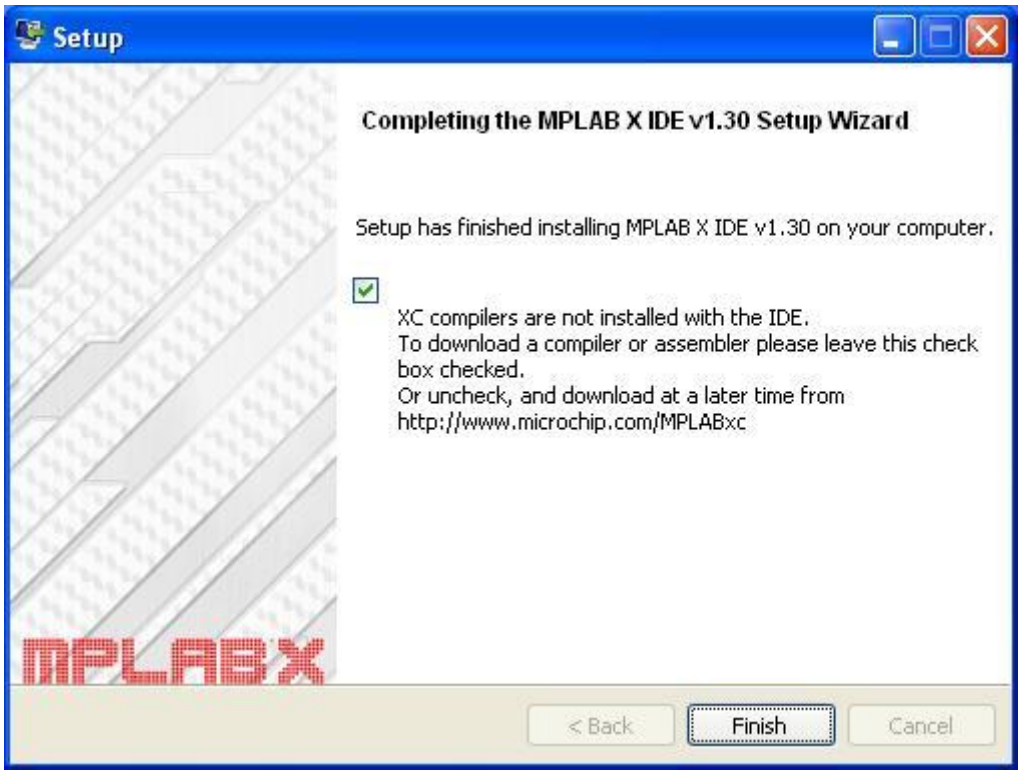




Merita attenzione invece la parte conclusiva, laddove compare un messaggio di di avviso per quanto concerne l'utilizzo dei programmer/debugger. Qualora si abbia installato anche MPLAB 8.xx è necessario eseguire un aggiornamento dei driver degli strumenti di sviluppo (come ad esempio PICKit3 e ICD3) alla versione di MPLAB X. Il passaggio al contrario è altrettanto possibile.



Infine, prima della conclusione, è possibile scaricare i compilatori XC; se già li si dispone oppure se non si è interessati, è sufficiente togliere la spunta e clicare su "Finish".

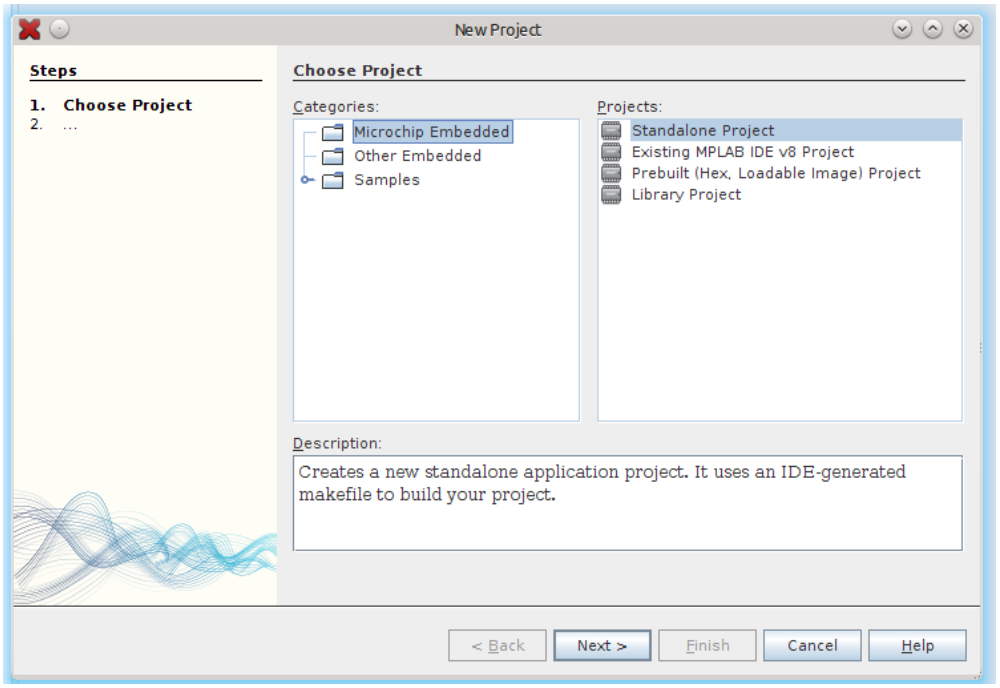


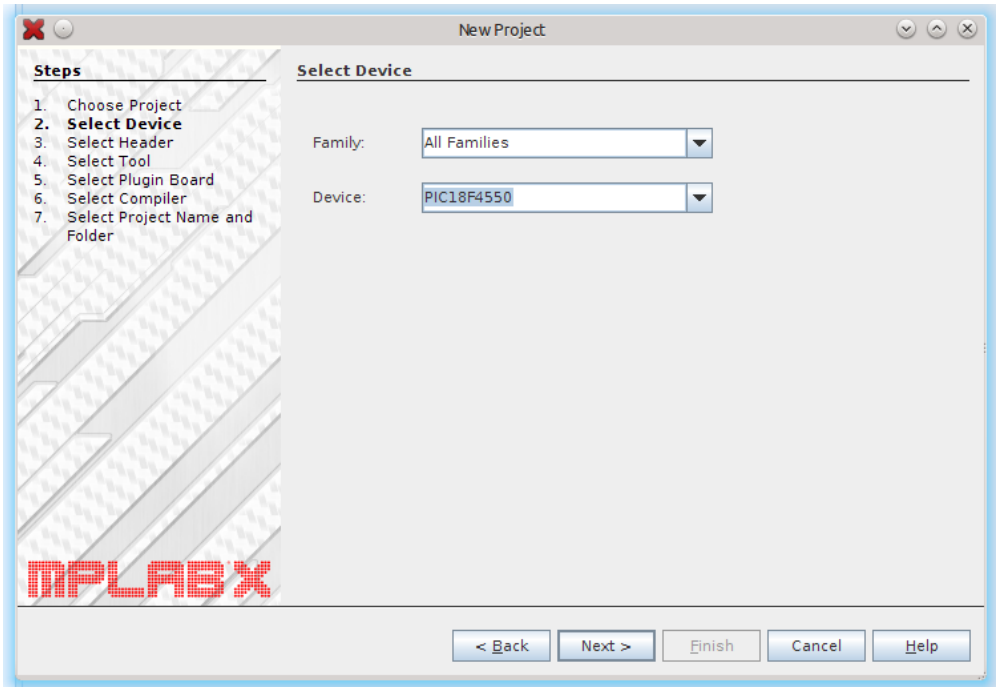
Le novità di MPLAB X

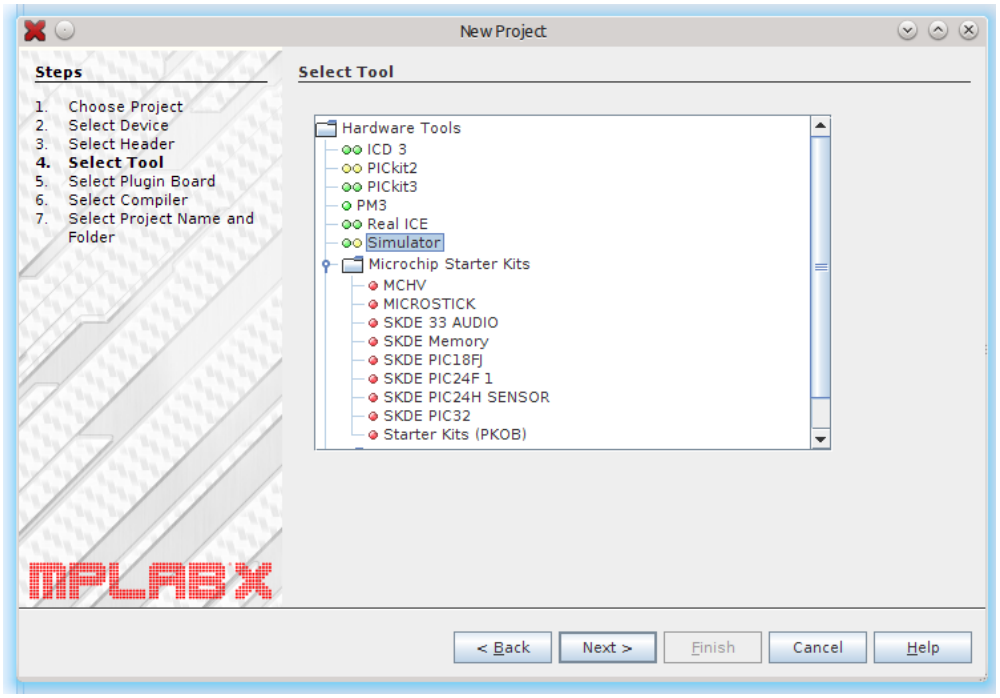
Non è mia intenzione descrivere tutte le funzionalità di MPLAB X; si rimanda alla lettura della **guida utente** [2], senz'altro più attendibile e completa di questo semplice articolo introduttivo.
La novità più evidente è senz'altro un diverso aspetto dell'IDE rispetto alla versione 8.xx. Un altro aspetto molto importante è che il progetto che si realizza non solo è legato al compilatore che si intende utilizzare ma anche al programmer/debugger.
Microchip ha pensato bene di mantenere una certa compatibilità con i progetti svolti con MPLAB 8.xx, progetti che si possono importare e convertire per il nuovo IDE, mantenendo intatte però le prerogative di ciascun ambiente. In sostanza si può lavorare su uno stesso progetto, con l'IDE X o 8.xx. L'aggiornamento dei sorgenti non preclude la visualizzazione degli stessi nei differenti IDE.
Anche con MPLAB X i compilatori di terze parti sono supportati e, udite udite, anche SDCC [3] diventa integrabile nell'IDE. Questo aspetto, che con MPLAB 8.xx non era possibile, permette di utilizzare anche questo compilatore in un unico ambiente di sviluppo.

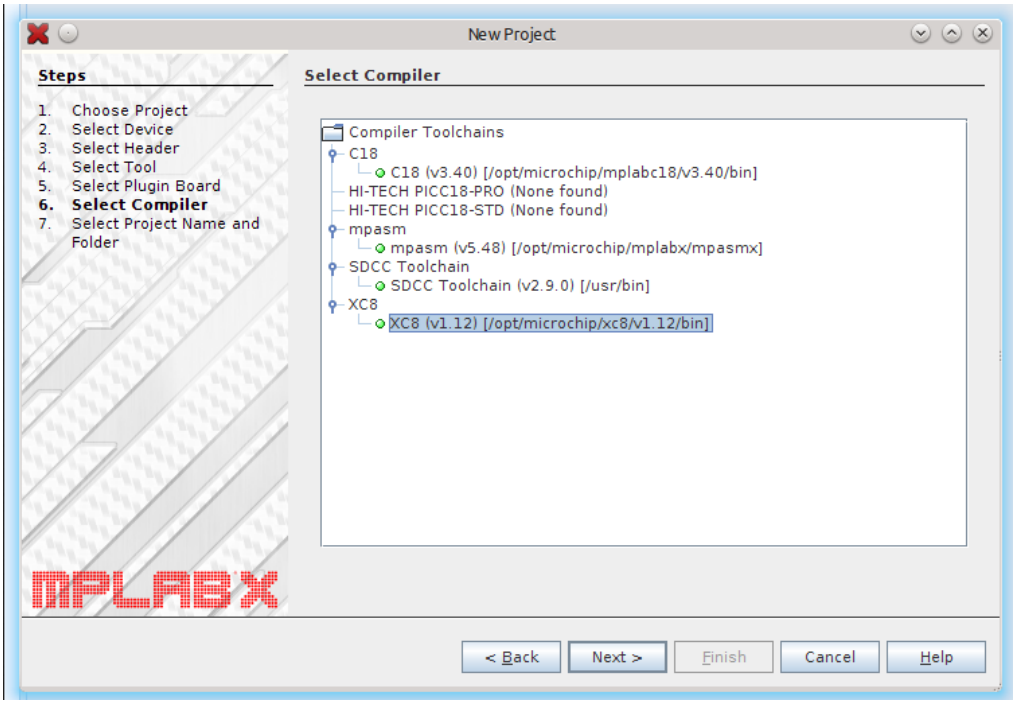
Finalmente il codice!

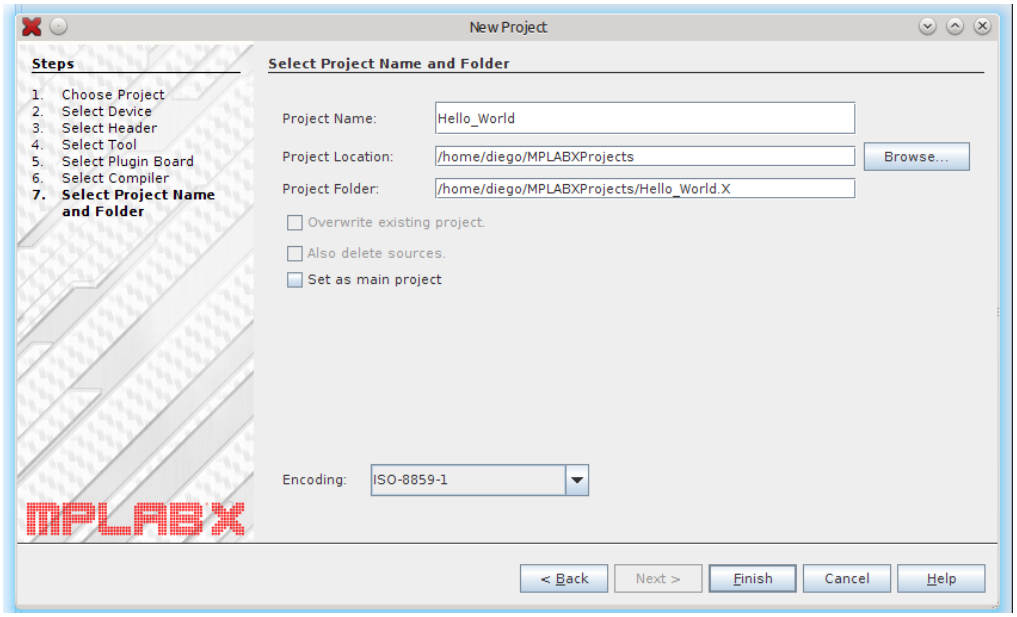
È arrivato finalmente il momento di creare un primo progetto con MPLAB X e mettere mano al codice. In pratica realizzeremo quello che è considerato, nel mondo dei microcontrollori, l'equivalente di ciò che in programmazione è un hello world, ovvero far lampeggiare un led. Tuttavia il led non lampeggerà da solo ma lo azioneremo con un pulsante, anzi i led azionati con pulsante saranno due.
Allora, avviamo MPLAB X e dal menù "File" clicchiamo su "New Projects…". Si aprirà la schermata mostrata in figura 4. Selezioniamo "Microchip Embedded" e quindi sulla destra "Standalone Project". Nella schermata successiva (figura 5) scriviamo il nome del microcontrollore utilizzato, ovvero PIC18F2550, nella sezione "Device" e andiamo avanti. Nella prossima schermata (figura 6) selezioniamo la voce "Simulator", in quanto non possediamo nessun debugger stand alone e anche perché in queste guide faremo esclusivamente uso del simulatore software. Al passo seguente dobbiamo scegliere il compilatore da utilizzare. Verranno mostrati tutti i compilatori installati sul proprio sistema compatibili con il PIC selezionato nella fase precedente, ognuno con rispettiva versione e directory di installazione. Scegliamo XC8 (figura 7) e proseguiamo. Nella schermata successiva (figura 8) immettiamo il nome del progetto nel campo "Project Name" e la cartella in cui salvarlo in "Project Location". Consiglio invece di non modificare il nome del campo "Project Folder", anche se ciò non crea problemi. Infatti l'ambiente di sviluppo tende ad aggiungere il suffisso .X alla cartella del progetto. Se lo desideriamo possiamo mettere la spunta su "Set as main projects", che rende attivo il progetto appena creato all'interno di MPLAB. Infatti MPLAB X permette di avere più progetti aperti contemporaneamente, ma solo uno di essi può essere attivo. Se non abbiamo altri progetti aperti quello creato sarà attivo per default. Clicchiamo infine su Finish.


New Project
Steps
1. Choose Project
2. ...
Choose Project
Categories:
Microchip Embedded
Other Embedded
Samples
Projects:
Standalone Project
Existing MPLAB IDE v8 Project
Prebuilt (Hex, Loadable Image) Project
Library Project
Description:
Creates a new standalone application project. It uses an IDE-generated makefile to build your project.
< Back
Next >
Finish
Cancel
Help

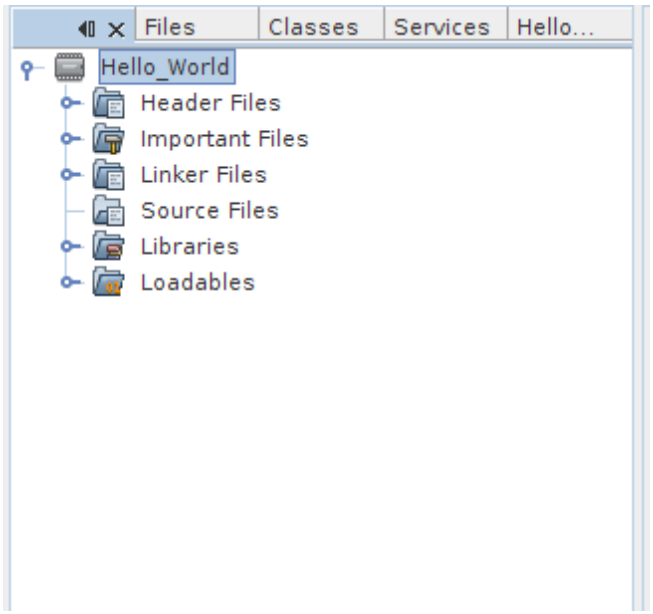

New Project
Steps
1. Choose Project
2. Select Device
3. Select Header
4. Select Tool
5. Select Plugin Board
6. Select Compiler
7. Select Project Name and Folder
Select Device
Family:
All Families
Device:
PIC18F4550
MPLABX
< Back
Next >
Finish
Cancel
Help


New Project
Steps
1. Choose Project
2. Select Device
3. Select Header
4. Select Tool
5. Select Plugin Board
6. Select Compiler
7. Select Project Name and Folder
Select Tool
Hardware Tools
ICD 3
PICkit2
PICkit3
PM3
Real ICE
Simulator
Microchip Starter Kits
MCHV
MICROSTICK
SKDE 33 AUDIO
SKDE Memory
SKDE PIC18FJ
SKDE PIC24F 1
SKDE PIC24H SENSOR
SKDE PIC32
Starter Kits (PKOB)
MPLAB X
< Back
Next >
Finish
Cancel
Help

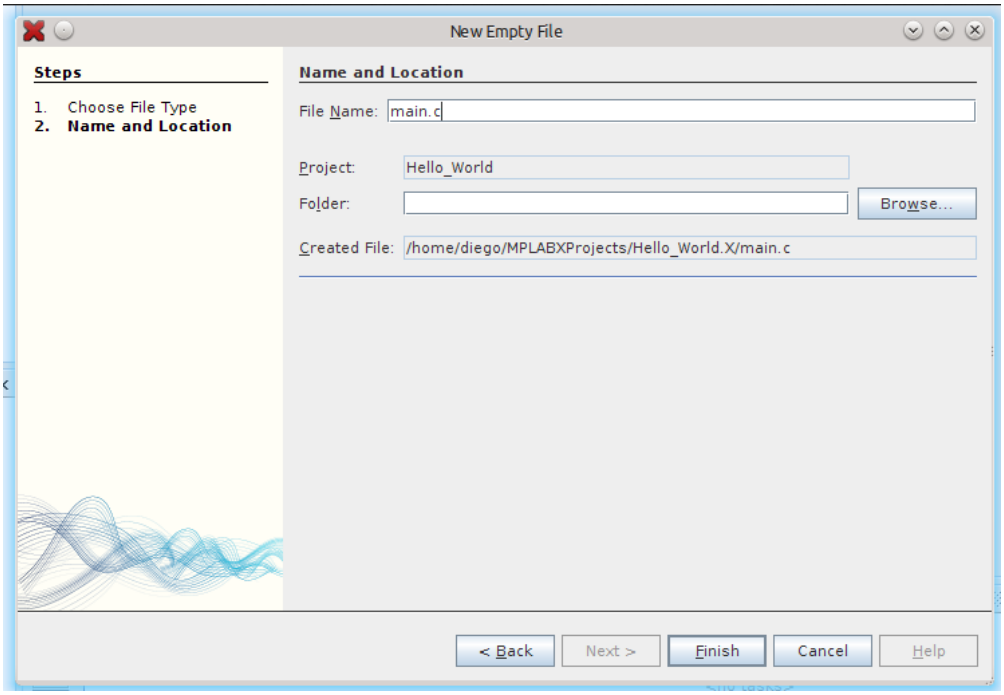

New Project
Steps
1. Choose Project
2. Select Device
3. Select Header
4. Select Tool
5. Select Plugin Board
6. Select Compiler
7. Select Project Name and Folder
Select Compiler
Compiler Toolchains
C18
C18 (v3.40) [/opt/microchip/mplabc18/v3.40/bin]
HI-TECH PICC18-PRO (None found)
HI-TECH PICC18-STD (None found)
mpasm
mpasm (v5.48) [/opt/microchip/mplabx/mpasmx]
SDCC Toolchain
SDCC Toolchain (v2.9.0) [/usr/bin]
XC8
XC8 (v1.12) [/opt/microchip/xc8/v1.12/bin]
MPLAB X
< Back
Next >
Finish
Cancel
Help



La creazione del progetto è completata e potremo vedere la sua struttura nella barra a sinistra (figura 9), con diverse cartelle ancora vuote. La cartella Header Files è preposta al contenimento dei file di intestazione del C, la cartella Source Files per i file sorgente, la cartella Libraries per i file di libreria e così via.

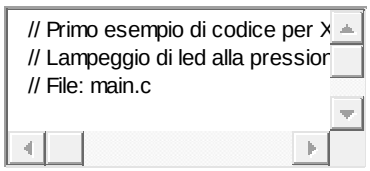


La prima cosa che dobbiamo fare è aggiungere il file sorgente che conterrà la funzione main. Quindi clicchiamo con il tasto destro sulla cartella Source Files, quindi “New”->”Empty File…”. Nella schermata che si aprirà immettiamo il nome del file sorgente, ad esempio chiamiamolo main.c (figura 10), quindi clicchiamo su Finish.



Il file vuoto verrà aperto nell'editor: incolliamo al suo interno il codice riportato di seguito e salviamo il progetto cliccando su "File"->"Salve All" o sull'apposita icona nella toolbar.

C




```
1  // Primo esempio di codice per XC8:
2  // Lampeggio di led alla pressione del corrispondente pulsante
3  // File: main.c
4
5  #include <xc.h>                 //Header file generico
6
7  void main(void) {
8
9      TRISA = 0xFF;               //Imposto tutti i pin di PORTA come input
10     TRISB = 0xFF;               //Imposto tutti i pin di PORTB come input
11     TRISC = 0xFF;               //Imposto tutti i pin di PORTC come input
```

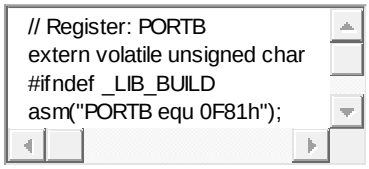
```
12    TRISD = 0xFF;                 //Imposto tutti i pin di PORTD come input
13    TRISE = 0xFF;                 //Imposto tutti i pin di PORTE come input
14
15    TRISBbits.TRISB2 = 0;         //Imposto il pin RB2 come output
16    TRISBbits.TRISB3 = 0;         //Imposto il pin RB3 come output
17
18    while (1) {                   //Ciclo infinito
19
20        //Led su RB2 associato al pulsante su RB5 (attivo basso)
21        if (PORTBbits.RB5 == 0) {
22            PORTBbits.RB2 = 1;
23        } else {
24            PORTBbits.RB2 = 0;
25        }
26
27        //Led su RB3 associato al pulsante su RB4 (attivo basso)
28        if (PORTBbits.RB4 == 0) {
29            PORTBbits.RB3 = 1;
30        } else {
31            PORTBbits.RB3 = 0;
32        }
33
34    }
35
36 }
```

Analizziamo il codice riga per riga. La direttiva

```
#include <xc.h> //Header file generico
```

aggiunge nel sorgente il file d'intestazione generico xc.h, che va incluso sempre all'inizio di ogni file facente parte del progetto. Questo file in pratica richiama, attraverso delle macro, il file d'intestazione specifico del dispositivo utilizzato, il quale contiene tutte le definizioni delle porte di I/O. Infatti le varie porte del microcontrollore sono mappate in memoria e, includendo questo file, vi si può accedere usando delle normali variabili C. Possiamo aprire questo file e vedere come è strutturato. Si tratta del file 18f2550.h, contenuto nella cartella include del compilatore. Consideriamo ad esempio la definizione della PORTB:

C




```
1   // Register: PORTB
2   extern volatile unsigned char       PORTB       @ 0xF81;
3   #ifndef _LIB_BUILD
4   asm("PORTB equ 0F81h");
5   #endif
6   // bitfield definitions
7   typedef union {
8       struct {
9           unsigned RB0            :1;
10          unsigned RB1            :1;
11          unsigned RB2            :1;
12          unsigned RB3            :1;
13          unsigned RB4            :1;
14          unsigned RB5            :1;
15          unsigned RB6            :1;
16          unsigned RB7            :1;
17      };
18      struct {
19          unsigned INT0           :1;
```

```
20      unsigned INT1           :1;
21      unsigned INT2           :1;
22      unsigned                :2;
23      unsigned PGM            :1;
24      unsigned PGC            :1;
25      unsigned PGD            :1;
26    };
27    struct {
28      unsigned                :3;
29      unsigned CCP2_PA2       :1;
30    };
31 } PORTBbits_t;
32 extern volatile PORTBbits_t PORTBbits @ 0xF81;
```

Vediamo che viene definita una variabile di tipo extern volatile unsigned char di nome PORTB, mappata all'indirizzo 0xF81. Quindi per scrivere ad esempio il valore 1 su RB0 (pin 33) di PORTB basterà effettuare l'assegnazione PORTB = 0x01, oppure PORTB = 0b00000001, dove il prefisso 0x davanti ad un valore numerico sta a significare che esso è in esadecimale, mentre 0b indica il formato binario; se non è riportato alcun prefisso il numero è inteso in decimale.
Notiamo che alla fine del codice viene definita un'altra variabile denominata PORTBbits, mappata allo stesso indirizzo. Tale variabile permette di accedere ai singoli bit della porta grazie all'uso di una union, ovvero una dichiarativa che consente di ridefinire una stessa area dati con nomi differenti. Quindi possiamo scrivere 1 su RB0 usando PORTBbits.RB0 = 1. Avremmo potuto anche scrivere PORTBbits.INT0 = 1, in quanto notiamo dal codice che RB0 viene ridefinita anche con il nome INT0. Ciò è dovuto al fatto che le porte hanno più funzionalità, quindi il sebbene il risultato sia lo stesso, si consiglia di scegliere il nome più appropriato a seconda del contesto (ad esempio RB0 se la funzione è di tipo digitale, come accendere un led, INT0 se si sta usando tale pin come una sorgente di interrupt esterno).
Ritornando al nostro hello world, segue la funzione main, che non prende alcun parametro e non restituisce alcun valore. In effetti non c'è un sistema operativo sul nostro dispositivo a cui restituire un valore di controllo, per questo la funzione è stata dichiarata come void.
La prima cosa che bisogna fare nel main è stabilire quali porte funzioneranno come input e quali come output. Questo è possibile farlo attraverso i registri TRIS. Nel nostro microcontrollore ne abbiamo 5 legati ai rispettivi registri PORT. Anche se non necessario, perché si tratta del comportamento di default, quello che facciamo in questo frammento di codice

```
TRISA = 0xFF; //Imposto tutti i pin di PORTA come input
TRISB = 0xFF; //Imposto tutti i pin di PORTB come input
TRISC = 0xFF; //Imposto tutti i pin di PORTC come input
TRISD = 0xFF; //Imposto tutti i pin di PORTD come input
TRISE = 0xFF; //Imposto tutti i pin di PORTE come input
```

è impostare tutte le porte del PIC come input (0 sta per output e 1 per input, è una convenzione). Dopodiché andiamo a settare come output le porte sulle quali abbiamo collegato i due led, agendo sui singoli bit:

```
TRISBbits.TRISB2 = 0; //Imposto il pin RB2 come output
TRISBbits.TRISB3 = 0; //Imposto il pin RB3 come output
```

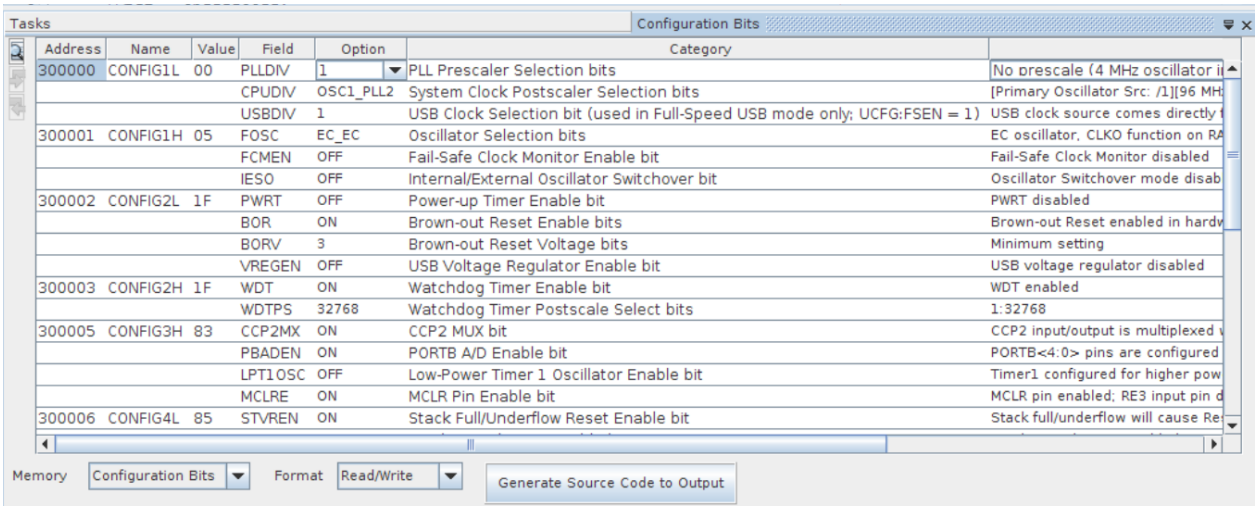
Segue poi un ciclo infinito, all'interno del quale si va ad eseguire il programma principale. Infatti il microcontrollore è progettato per eseguire una dopo l'altra tutte le istruzioni in memoria; quando arriva alla fine il Program Counter si azzera e riparte dall'inizio. Per fare in modo che il PIC ripeta sempre le stesse operazioni si inserisce un while(1), oppure un for(;;;) che è lo stesso. All'interno del ciclo si va a verificare se il valore logico sul pin RB5 è basso (condizione di pulsante chiuso), in caso affermativo si pone a 1 il pin RB2, ovvero si accende il led ad esso collegato. Analogo controllo viene effettuato sul tasto su RB4 per accendere o spegnere il led su RB3. Il PIC continuerà ad effettuare queste controlli all'infinito.

Esempio programma per lampeggio LED:

```
// Primo esempio di codice per xc8:

// lampeggio di un led alla pressione di un pulsante

// File: main.c



#include <xc.h>

int i;

void main (void)    {

   TRISA = 0xFF;

   TRISB = 0xFF;

   TRISC = 0xFF;



   TRISBbits.TRISB2 = 0;

   TRISBbits.TRISB3 = 0;



   while (1)   {

      // Led su RB2 associato al pulsante su RB5 ( attivo basso)
```

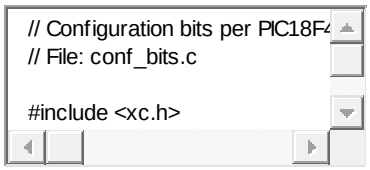
```
        if (PORTBbits.RB4 == 0) {
            PORTBbits.RB2 = 1;
            for(i = 0;i <= 500; i++)
            _delay(100000);



        } else {PORTBbits.RB2 = 0;
    }
        // Led su RB3 associato al pulsante su RB4 ( attivo basso)
        if (PORTBbits.RB5 == 0) {
            PORTBbits.RB3 = 1;
        } else {PORTBbits.RB3 = 0;
    }
}
}
```

Prima di compilare il programma dobbiamo impostare i registri di configurazione del PIC, i cosiddetti Configuration Bits. In pratica si tratta di registri del PIC che consentono di settare le diverse opzioni dell'hardware, ad esempio i prescaler per la divisione del clock, abilitare o meno una periferica, proteggere in scrittura determinate aree di memoria, ecc.
In MPLAB 8 esisteva la possibilità di impostare tutti questi parametri in una finestra dell'ambiente di sviluppo, senza inserirli nel codice. In MPLAB X i configuration bits devono per forza essere inseriti nel codice. Ci viene in aiuto un tool grafico, il quale, una volta settati i registri che ci interessano, genererà per noi il file sorgente con i configuration bits. Tale tool è accessibile da "Window"-> "PIC Memory Views"-> "Configuration Bits" (figura 11).



Come notiamo dalla figura, il nome del parametro di configurazione è riportato nel campo “Field”, mentre attraverso dei menù a tendina sotto la voce “Option” sono selezionabili le varie possibilità di scelta per ogni parametro. È riportata anche una descrizione dei vari parametri e delle opzioni sotto le voci “Category” e “Setting” rispettivamente. Una volta impostati tutti i configuration bits si può generare il file sorgente che li contiene cliccando su “Generate Source Code to Output”. Per evitarvi di impostare i configuration bits uno ad uno riporto io il codice, che dovete copiare ed incollare in un nuovo file da aggiungere al progetto (sapete come fare):

C




```
1  // Configuration bits per PIC18F4550
2  // File: conf_bits.c
3
4  #include <xc.h>
5
6  // CONFIG1L
7  #pragma config PLLDIV = 5
8  #pragma config CPUDIV = OSC1_PLL2
9  #pragma config USBDIV = 2
10
11 // CONFIG1H
```

```
12 #pragma config FOSC = HSPLL_HS
13 #pragma config FCMEN = OFF
14 #pragma config IESO = OFF
15
16 // CONFIG2L
17 #pragma config PWRT = OFF
18 #pragma config BOR = ON
19 #pragma config BORV = 3
20 #pragma config VREGEN = ON
21
22 // CONFIG2H
23 #pragma config WDT = OFF
24 #pragma config WDTPS = 32768
25
26 // CONFIG3H
27 #pragma config CCP2MX = ON
28 #pragma config PBADEN = OFF
29 #pragma config LPT1OSC = OFF
30 #pragma config MCLRE = ON
31
32 // CONFIG4L
33 #pragma config STVREN = ON
34 #pragma config LVP = OFF
35 #pragma config ICPRT = OFF
36 #pragma config XINST = OFF
37
38 // CONFIG5L
```

```
39 #pragma config CP0 = OFF
40 #pragma config CP1 = OFF
41 #pragma config CP2 = OFF
42 #pragma config CP3 = OFF
43
44 // CONFIG5H
45 #pragma config CPB = OFF
46 #pragma config CPD = OFF
47
48 // CONFIG6L
49 #pragma config WRT0 = OFF
50 #pragma config WRT1 = OFF
51 #pragma config WRT2 = OFF
52 #pragma config WRT3 = OFF
53
54 // CONFIG6H
55 #pragma config WRTC = OFF
56 #pragma config WRTB = OFF
57 #pragma config WRTD = OFF
58
59 // CONFIG7L
60 #pragma config EBTR0 = OFF
61 #pragma config EBTR1 = OFF
62 #pragma config EBTR2 = OFF
63 #pragma config EBTR3 = OFF
64
65 // CONFIG7H
```

66 #pragma config EBTRB = OFF

Notiamo subito che i configuration bits si specificano con la direttiva #pragma config. In realtà c'è anche un altro modo per specificarli, che però non vediamo per evitare confusione. Quella che stiamo usando è la modalità ereditata dal C18 per il settaggio di tali parametri, che personalmente preferisco. Gran parte dei configuration bits riportati sono al loro valore di default, quindi avremmo potuto anche non aggiungerli. Analizziamone qualcuno. Importanti sono quelli per la configurazione del clock, ad esempio il seguente

```
#pragma config PLLDIV = 5
```

che imposta a 5 il il fattore di divisione del prescaler a monte del PLL (vedi figura 3). Poi abbiamo

```
#pragma config CPUDIV = OSC1_PLL2
```

che divide per due l'uscita da 96 MHz del PLL. Il clock dell'USB viene prelevato dall'uscita del PLL e diviso per due con quest'altra riga di codice:

```
#pragma config USBDIV = 2
```

Infine con la seguente

```
#pragma config FOSC = HSPLL_HS
```

si imposta il generatore del clock in modalità HS con PLL abilitato. Altri configuration bits degni di nota sono i seguenti:

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

Il primo disabilita gli ingressi analogici su PORTB, il secondo disabilita la programmazione LVP, permettendo di usare il pin 38 (PGM) come I/O general purpose. Per tutti gli altri rimando al datasheet del PIC18F2550.

Esempio di configurazione per il PIC18F2550:

// Configuration bits per PIC18F4550

// File: conf_bits.c

#include <xc.h>

// CONFIG1L

```
#pragma config PLLDIV = 5   // prescaler divide per 5 a monte del PLL
#pragma config CPUDIV = OSC1_PLL2   // divide l'uscita del PLL da 96 MHz per due
#pragma config USBDIV = 2   // uscita del PLL viene divisa per due e usaata da USB

// CONFIG1H
#pragma config FOSC = HSPLL_HS  //imposta l'oscillatore in modalità HS co uso del PLL
#pragma config FCMEN = OFF  //disabilita il Fail-save monitor
#pragma config IESO = OFF   //disabilita la modalità Switchover

// CONFIG2L
#pragma config PWRT = OFF   //disabilita il Power-up timer
#pragma config BOR = ON //Brown-out reset hardware abilitato
#pragma config BORV = 3 //soglia del Brown-out reset impostata a 2.0 V
#pragma config VREGEN = ON  //abilita il generatore di 3.3V per USB

// CONFIG2H
#pragma config WDT = OFF    //disabilita il timer del Watchdog
#pragma config WDTPS = 32768    // imposta prescaler del timer del watchdog

// CONFIG3H
#pragma config CCP2MX = ON  //CCP2 su RC1
#pragma config PBADEN = OFF //disabilita gli ingressi analogici su PORTB
#pragma config LPT1OSC = OFF    //Timer1 configurato in modalità higher power
#pragma config MCLRE = ON   // pin MCLR abilitato (pin RE3 disabilitato)

// CONFIG4L
```

```
#pragma config STVREN = ON  // abilita reset di stack full

#pragma config LVP = OFF    //disabilita la programmazione LVP permettendo di usare il pin PGM come I/O

//#pragma config ICPRT = OFF  //disabilita pin ICSP alternativi (solo 18F4550)

#pragma config XINST = OFF  //disabilita set esteso delle istruzioni


// CONFIG5L

#pragma config CP0 = OFF    //code protectin blocco 0 disabilitato

#pragma config CP1 = OFF    //code protectin blocco 1 disabilitato

#pragma config CP2 = OFF    //code protectin blocco 2 disabilitato

#pragma config CP3 = OFF    //code protectin blocco 3 disabilitato


// CONFIG5H

#pragma config CPB = OFF    //code protectin blocco di boot disabilitato

#pragma config CPD = OFF    //code protectin eeprom disabilitato


// CONFIG6L

#pragma config WRT0 = OFF   //write protect block 0 disabilitato

#pragma config WRT1 = OFF   //write protect block 1 disabilitato

#pragma config WRT2 = OFF   //write protect block 2 disabilitato

#pragma config WRT3 = OFF   //write protect block 3 disabilitato


// CONFIG6H

#pragma config WRTC = OFF   //write protect configurations disabilitato

#pragma config WRTB = OFF   //write protect boot block disabilitato

#pragma config WRTD = OFF   //write protect eeprom disabilitato
```

// CONFIG7L

#pragma config EBTR0 = OFF //read protect table block 0 disabilitato

#pragma config EBTR1 = OFF //read protect table block 1 disabilitato

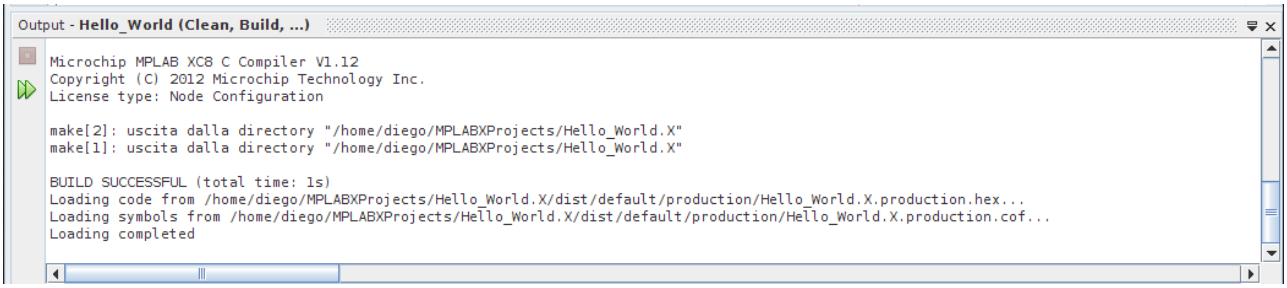
#pragma config EBTR2 = OFF //read protect table block 2 disabilitato

#pragma config EBTR3 = OFF //read protect table block 3 disabilitato

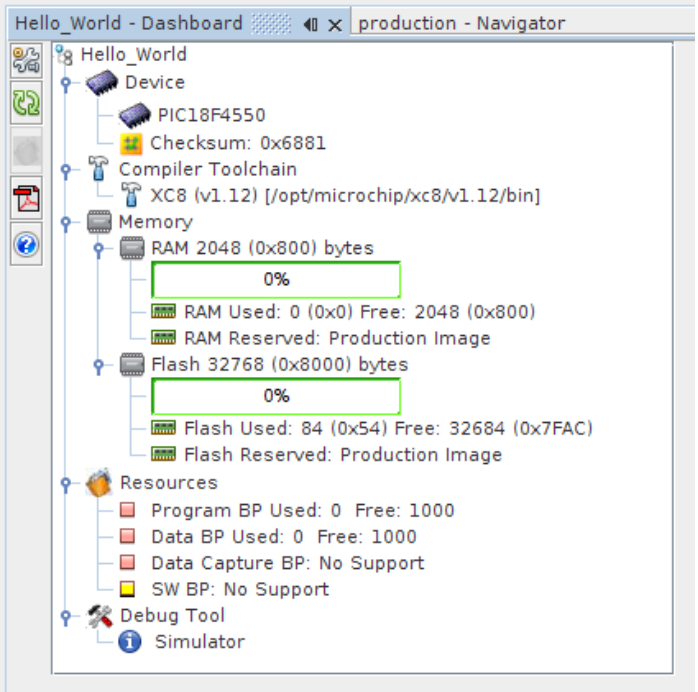
// CONFIG7H

#pragma config EBTRB = OFF //boot block non protetto dalla lettura de tabelle di altri blocchi

A questo punto compiliamo il progetto cliccando sull'apposita icona nella toolbar, oppure da menù con "Run"->"Build Project". Il compilatore redigerà tutti i messaggi nella finestra "Output", posizionata in basso nell'ambiente di sviluppo. Se non ci sono errori avremo un output come quello mostrato in figura 12.



Possiamo vedere l'assembly generato dalla compilazione accedendo al menù "Window"-> "Output"-> "Disassembly Listing File", dove possiamo notare, per ogni linea di codice in C la corrispondente "traduzione" in assembly. Cliccando su "Window"-> "Dashboard" invece l'ambiente ci apre una schermata con un riepilogo sulle memorie del PIC ed altri dettagli (figura 13). Il nostro PIC infatti ha due memorie: una memoria programma (Flash), non volatile e di sola lettura, nella quale è appunto memorizzato il codice e le costanti, e una memoria dati (RAM), volatile, nella quale sono memorizzate le variabili (detto in maniera grossolana). Notiamo che nel nostro caso la memoria dati è inutilizzata, perché non abbiamo dichiarato alcuna variabile.



La compilazione avrà generato il file HEX che possiamo usare per programmare il microcontrollore. Il file suddetto si trova nella cartella

```
dist/default/production
```

a partire dalla cartella principale del progetto e avrà lo stesso nome del progetto, con estensione .hex. In questa cartella si trovano anche altri file, in particolare il file .map generato dal linker, il listato assembly e così via.

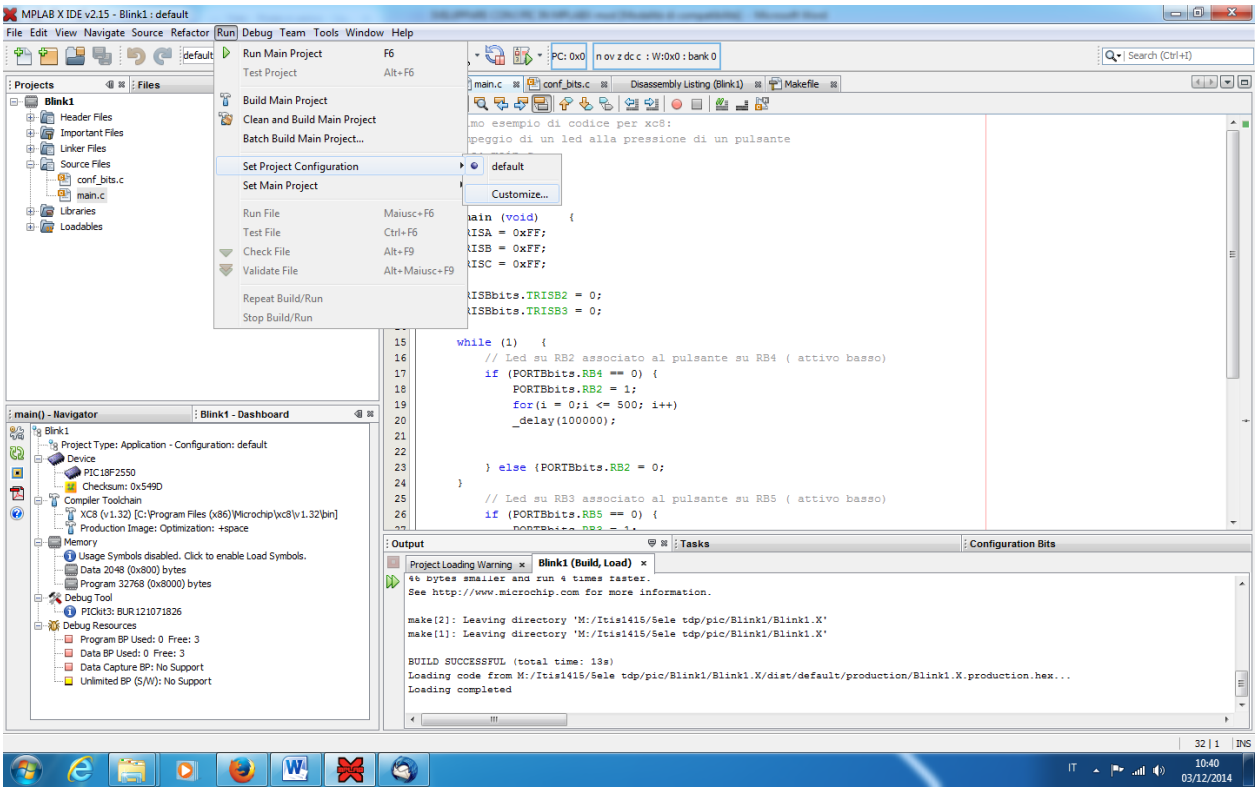
Programmare il PIC

Per programmare il microcontrollore ovviamente è necessario usare un programmatore (quello hardware, non il tizio seduto davanti al pc!). Per chi è alle prime armi consiglio di acquistare un PicKit3, costa poco (una trentina di euro, direttamente dallo store della Microchip) ed è affidabile. Per chi ha voglia di costruirselo da sé ci sono molti schemi di programmatori disponibili online. Il più diffuso è il JDM, che sfrutta la porta seriale per comunicare con il PC. Alcuni miei colleghi lo hanno testato in abbinamento al software PICPgm, disponibile sia per Windows che per Linux (per quest'ultimo è solo a riga di comando). Per maggiori informazioni consultate questo articolo: http://www.provalotu.com/featured/pic-programmazione/. Io personalmente uso il minipropic2, il cui schema è riportato a questo link: http://stor.altervista.org/pic/fcpic/fcpic.htm, in abbinamento al software winpic800 (disponibile solo per Windows), ma dovrebbe essere compatibile anche con Linux usando l'ambiente PikLab. È una bomba: veloce e non ha mai fallito una programmazione, solo che necessita della porta parallela. Se desiderate un programmatore con porta USB consiglio di realizzare l'Open Programmer, che trovate qui http://openprog.altervista.org/ e che è in fase di realizzazione nel mio "laboratorio segreto". Un altro programmatore su porta USB, che quasi sicuramente costruirò più in avanti, è il PICPgm USB Programmer: http://picpgm.picprojects.net/hardware.html#USB_PROGRAMMER, compatibile anch'esso con il software PICPgm. Questi sono i programmatori dei quali mi sono più interessato, ma ne esistono un'infinità.
Anche se, come ho ribadito fino alla noia, tra qualche lezione useremo il bootloader, il programmatore ci servirà comunque la prima volta per caricare il codice nel PIC. Quindi non esitate

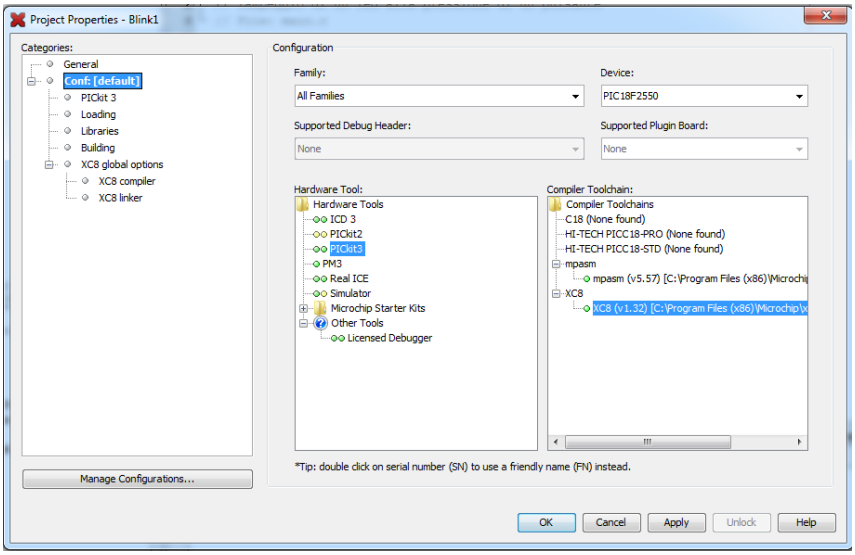
a comperare/costruire un programmatore, che è la prima cosa alla quale provvedere quando si decide di avere a che fare con i PIC.
Per questo articolo è tutto, nella prossima puntata vedremo qualcosa di più interessante, come la gestione degli interrupt, la generazione dei delay con il metodo waste time e scopriremo il simulatore di MPLABX.

Programmare con Pickit3 e MPLAB X

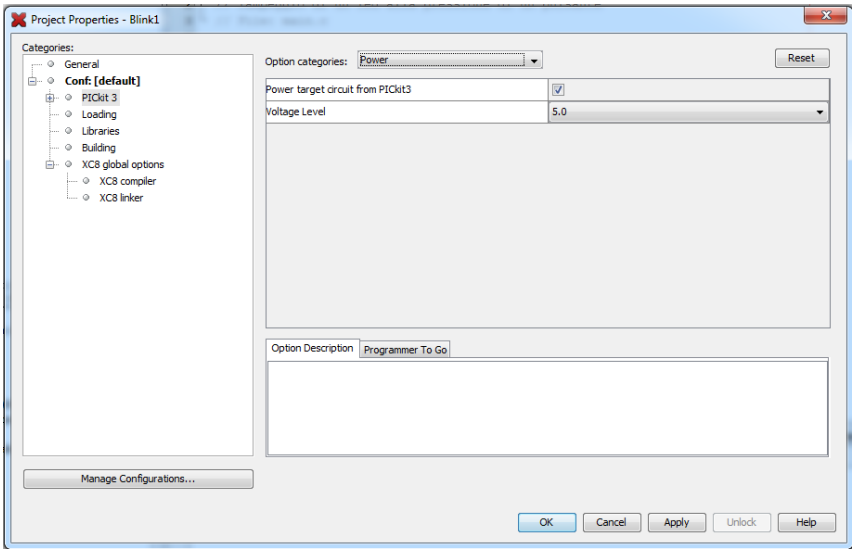
Prima bisogna configurare il progetto



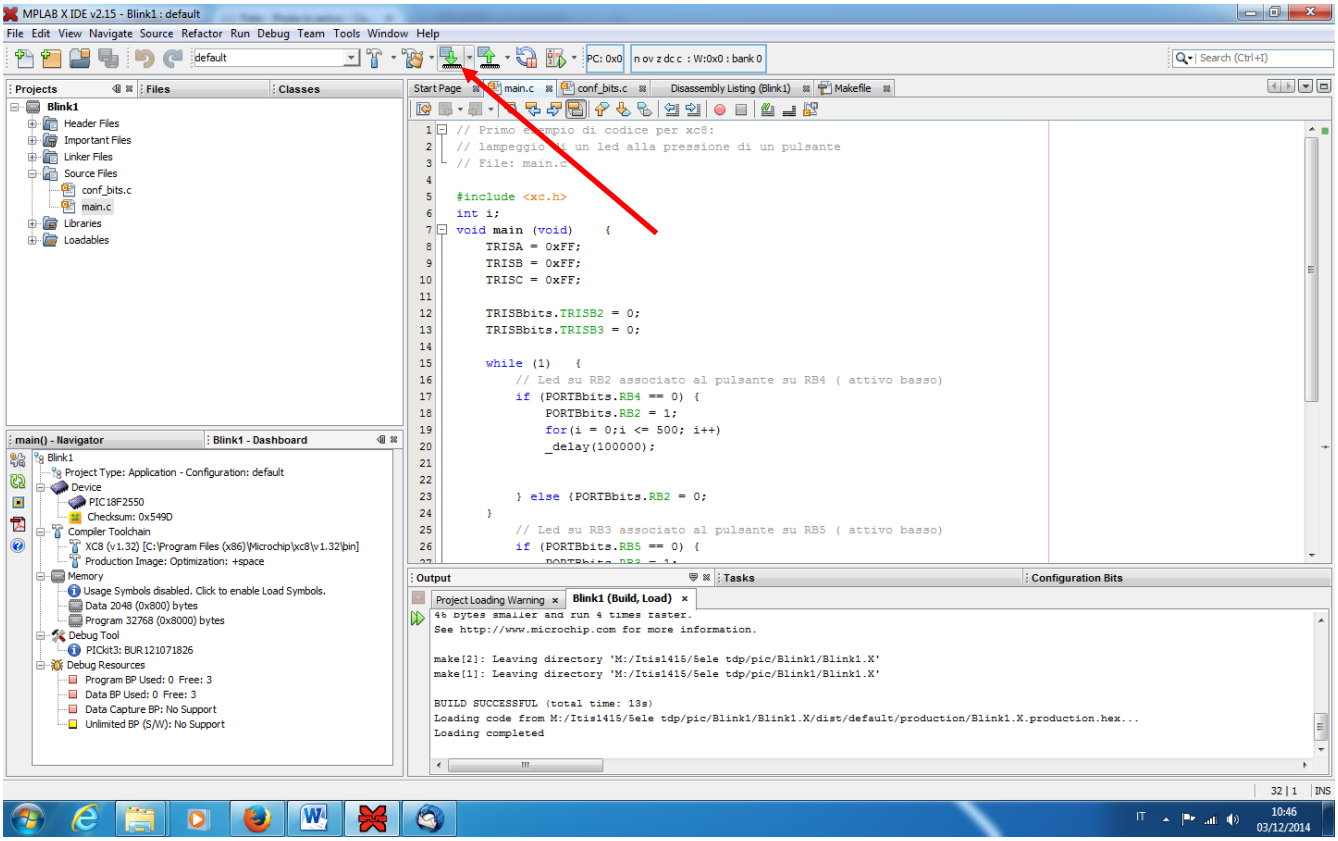
Settare Pickit3 come hardware

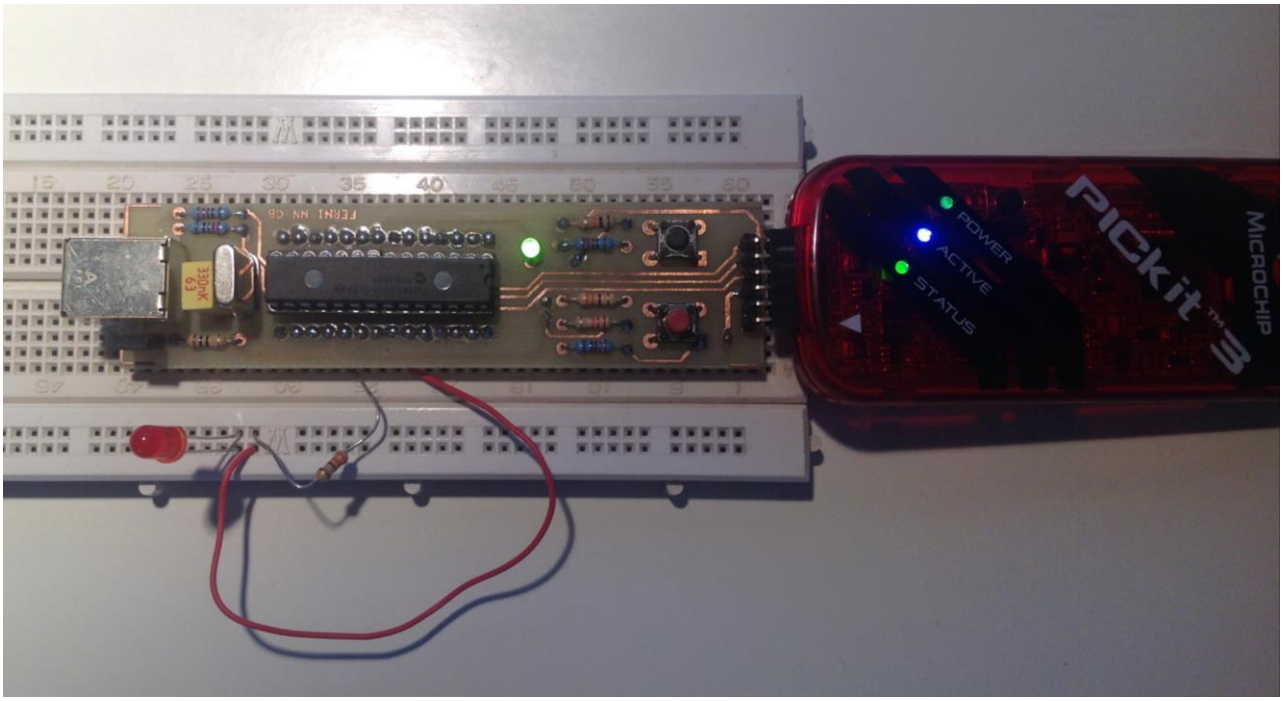


Abilitare l’alimentazione a 5V dal Pickit3



Usare il tasto Make and Program





I delay con il metodo waste time

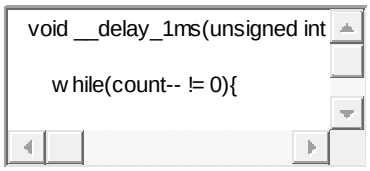
E se adesso, invece di azionare il led con la pressione di un tasto, volessimo farlo lampeggiare ad una determinata frequenza come potremmo fare? Basterà accendere il led nella prima metà del periodo e spegnerlo nell'altra metà. Questo richiede necessariamente l'inserimento di un ritardo tra l'accensione e lo spegnimento.
Tipicamente si adottano due approcci per generare ritardi con un microcontrollore. Il primo metodo consiste nell'inserire delle istruzioni che non fanno assolutamente nulla, dette NOP (No Operation), ognuna delle quali dura un ciclo di clock, oppure delle altre istruzioni che non producono risultati utili (ad esempio dei cicli a vuoto), che servono solo a far perdere tempo al microcontrollore. Tale metodo è detto waste time, ovvero tempo perso. Il vantaggio di questo approccio è che non impegna alcuna periferica del microcontrollore, tuttavia durante il ritardo il PIC non può svolgere nessun'altra operazione, deve necessariamente aspettare la fine del delay. Un secondo approccio è quello usare delle particolari periferiche del PIC, i timer, in pratica dei contatori a decremento. Il vantaggio è che poiché i timer lavorano indipendentemente dalla cpu, quest'ultima può essere impiegata per svolgere altri compiti. Noi useremo il primo approccio, mentre parleremo dei timer negli articoli che seguiranno.
Il compilatore XC8 possiede delle apposite funzioni per la generazione dei ritardi, che in pratica non fanno altro che inserire un certo numero di istruzioni NOP nel punto del codice in cui esse sono richiamate. Tali funzioni sono disponibili da subito includendo il file d'intestazione generico xc.h. In particolare abbiamo due funzioni: _delay() e _delay3(). Esse prendono come parametro un numero intero (unsigned long) che rappresenta il numero di istruzioni NOP da inserire nel codice. Nella delay 3, come s'intuisce, tale numero è moltiplicato per 3. Ad esempio, con una FCYC di 12 MHz la durata di una singola istruzione è di 83 ns, per cui la chiamata a funzione _delay(2) introduce tra l'istruzione precedente e quella successiva un delay di 166 ns. Tali funzioni sono delle funzioni inline, cioè vengono espanse nel codice al momento della compilazione; ciò per evitare ulteriori ritardi dovuti alla chiamata a funzione. Questa cosa tuttavia pone un limite al massimo argomento passabile alla funzione, e come vedremo, ciò costituisce per noi un problema che dovremo in un certo modo aggirare.

Generare un ritardo in termini della durata della singola istruzione è un po' scomodo, perciò il compilatore ci mette a disposizione altre due funzioni, __delay_us() e __delay_ms(), in cui questa volta il parametro passato alla funzione rappresenta la durata del ritardo, inteso rispettivamente in microsecondi e in millisecondi. Facciamo attenzione che queste funzioni sono precedute da un doppio underscore, mentre le delay semplici da uno singolo. Tali funzioni in pratica sfruttano la già citata funzione _delay() per creazione di ritardi di durata pari al tempo specificato. Per usarle è obbligatorio comunicare al compilatore la frequenza di clock del microcontrollore (FOSC) mediante la direttiva al preprocessore _XTAL_FREQ.
Come già accennato, la funzione _delay è una funzione inline, quindi c'è un limite al massimo argomento che possiamo passarle e che ho verificato essere, sul mio sistema, di 197120. Provate a passarle un valore superiore e vedrete che il compilatore genererà il seguente errore:

```
main.c:33: error: delay exceeds maximum limit of 197120 cycles
```

A frequenze elevate, usando la funzione __delay_ms (che come già detto sfrutta internamente la _delay) tale valore massimo viene rapidamente raggiunto, non permettendo di generare ritardi al di sopra di una certa durata. Ad esempio, con una frequenza dell'oscillatore di 48 MHz ho sperimentato che non è possibile andare oltre i 16 ms di ritardo. Il problema ovviamente si verifica anche per la funzione __delay_us, e in tal caso il ritardo massimo è di circa 16000 µs (sempre i soliti 16 ms); ma poiché al di sopra dei 1000 µs si va di solito ad usare la funzione __delay_ms, il problema in tal caso non si pone. Per aggirare questa limitazione sui millisecondi possiamo crearci una nostra funzione, che utilizza la __delay_us all'interno di un ciclo while, e che utilizzeremo al posto di quella fornitaci dal compilatore:

C

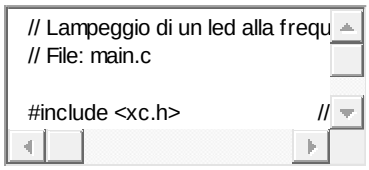



```
1 void __delay_1ms(unsigned int count){
2
3    while(count-- != 0){
4
5       __delay_us(1000);          //Ritardo di 1 ms
6
7    }
8
9 }
```

Tra i file di libreria dell'XC8 è disponibile anche una libreria di ritardi ereditata dal C18 (chi usa quest'ultimo la conosce bene), che si può utilizzare includendo il file d'intestazione delays.h.

Tuttavia tale libreria, almeno per quanto ho verificato personalmente, non funziona bene. Forse è stata testata con le versioni a pagamento del compilatore, che forniscono un codice più compatto (meno istruzioni, quindi ritardi minori), mentre con la nostra versione Free i ritardi sono tutti sballati.
Veniamo dunque al codice per far lampeggiare il led. Creiamo un nuovo progetto, come abbiamo imparato nella scorsa lezione, aggiungiamo ad esso il file con i configuration bits, che è sempre lo stesso, quindi possiamo copiarlo dal progetto precedente e un file vuoto all'interno del quale incolleremo il seguente codice:

C




```
1  // Lampeggio di un led alla frequenza di 2 Hz
2  // File: main.c
3
4  #include <xc.h>                  //Header file generico
5
6  #define _XTAL_FREQ 48000000          //Specifico la frequenza dell'oscillatore
7
8  //Prototipi di funzione
9  void __delay_1ms(unsigned int count);
10
11 void main(void) {
12
13    TRISA = 0xFF;                 //Imposto tutti i pin di PORTA come input
14    TRISB = 0xFF;                 //Imposto tutti i pin di PORTB come input
15    TRISC = 0xFF;                 //Imposto tutti i pin di PORTC come input
16    TRISD = 0xFF;                 //Imposto tutti i pin di PORTD come input
17    TRISE = 0xFF;                 //Imposto tutti i pin di PORTE come input
18
19    TRISBbits.TRISB2 = 0;           //Imposto il pin RB2 come output
```

```
20    TRISBbits.TRISB3 = 0;          //Imposto il pin RB3 come output
21
22    while (1) {                    //Ciclo infinito
23
24       PORTBbits.RB2 = 1;          //Led acceso
25       __delay_1ms(250);           //Ritardo di 250 ms
26       PORTBbits.RB2 = 0;          //Led spento
27       __delay_1ms(250);           //Ritardo di 250 ms
28
29    }
30
31 }
32
33 void __delay_1ms(unsigned int count){
34
35    while(count-- != 0){
36
37       __delay_us(1000);           //Ritardo di 1 ms
38
39    }
40
41 }
```

Tale programma ha la stessa impostazione di quello della lezione precedente, per cui richiede poche precisazioni. La prima riguarda la riga

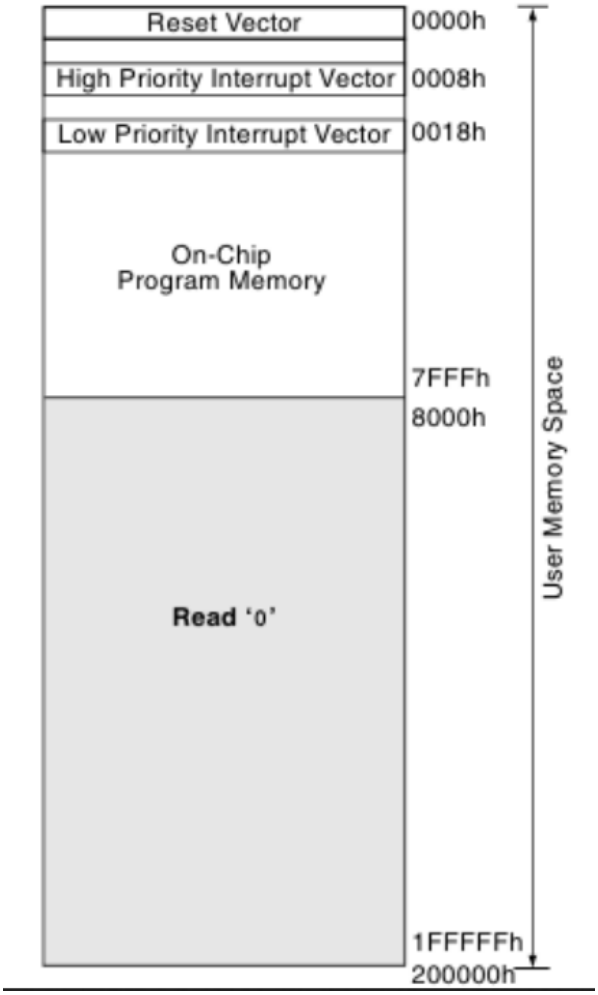
```
#define _XTAL_FREQ 48000000 //Specifico la frequenza dell'oscillatore
```

nella quale andiamo a specificare la frequenza, in Hz, dell'oscillatore utilizzato, direttiva richiesta dalle funzioni delay dell'XC8. Il compilatore userà questo valore per calcolare quanti cicli di clock sono necessari per ottenere il ritardo richiesto. Notiamo poi la presenza del prototipo della funzione __delay_1ms da noi definita, necessario perché è stata posta dopo il main. Infine vediamo come nel

ciclo si vada a mettere alternativamente a 1 e a 0 il pin RB2 sul quale è collegato il led, inserendo tra un'istruzione e l'altra un ritardo di 250 ms; quindi la frequenza di lampeggio è di 2 Hz

La gestione degli interrupt

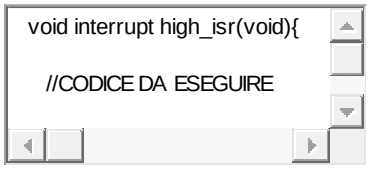
Per come abbiamo strutturato il programma precedente, il nostro PIC sarà impegnato per il 100% del suo tempo ad accendere e spegnere un led, quindi apparentemente non potrà fare null'altro. In realtà i microcontrollori vengono utilizzati per svolgere più compiti contemporaneamente. Immaginiamo di voler accendere e spegnere un led, ma nello stesso tempo avere la possibilità di accendere l'altro led con la pressione di un pulsante. Possiamo pensare di inserire all'inizio del ciclo un controllo sul tasto interessato, che verifica appunto se l'ingresso del PIC sul quale è collegato il tasto è al valore alto o al valore basso e agire sul corrispondete led di conseguenza (esattamente come fatto nella scorsa lezione). Il microcontrollore effettuerà sempre un'operazione alla volta, ovvero prima effettuerà il controllo e quindi accenderà o spegnerà il led azionato manualmente, poi accenderà il led automatico e poi lo spegnerà, ma queste azioni verranno fatte in maniera così veloce da dare l'idea che siano contemporanee (è il segreto del multitasking!). In pratica la tecnica secondo cui si va di volta in volta ad interrogare una particolare periferica (nel nostro caso un tasto) e agire di conseguenza è detta polling. Quando si usano i delay con il metodo waste time questa tecnica può creare dei problemi. Nel nostro caso se la pressione del tasto avviene durante il delay, l'accensione del led azionato manualmente avverrà solo dopo il termine di tale delay, perché appunto la cpu è impegnata ad "eseguire" il delay.
La soluzione a ciò è utilizzare gli interrupt. Un interrupt, come dice la parola, è un segnale di interruzione che può provenire da una particolare periferica interna al PIC, o dall'esterno. Tale segnale va appunto ad interrompere il core del microcontrollore, che è costretto a fermare quello che stava facendo e a, come si dice, "servire" l'interruzione, ovvero eseguire un codice specifico per quella linea di interruzione, detto Interrupt Service Routine (ISR). Prima di servire l'interruzione il microcontrollore dovrà salvare tutti i suoi registri interni, in maniera tale da ripristinarli una volta finito di servire l'interruzione e riprendere il flusso del programma principale. Il PIC18F4550 possiede due livelli di interruzione: interruzioni ad alta priorità ed interruzioni a bassa priorità. Le interruzioni a bassa priorità possono interrompere il normale flusso del programma, mentre le interruzioni ad alta priorità possono interrompere oltre al programma principale, anche il codice associato alle interruzioni a bassa priorità. Solo il segnale di reset può interrompere le interruzioni ad alta priorità.
In pratica, quando arriva un segnale di interrupt, il program counter (PC) del microcontrollore viene fatto puntare ad una particolare locazione di memoria, a partire dalla quale è contenuta la ISR. L'insieme delle locazioni di memoria destinato al codice per gli interrupt è detto vettore delle interruzioni, posizionato all'indirizzo 0x8 per gli interrupt ad alta priorità e all'indirizzo 0x18 per quelli a bassa priorità, mentre ovviamente il reset è posizionato all'indirizzo 0, come vediamo dalla figura 1, che riporta uno schema della memoria programma del nostro PIC. Notiamo che, sebbene il nostro PIC abbia 32 KB (0x7FFF) di flash, il PC è di 21 bit, quindi può indirizzare fino a 2 MB (0x1FFFFF), per cui le locazioni di memoria oltre alla 0x7FFF sono lette come 0 (in quanto non esistenti).



La gestione degli interrupt all'interno dei PIC è resa possibile mediante la manipolazione dei bit di alcuni registri. Andare a modificare i bit dei registri interni del microcontrollore è una pratica comune alla quale ci dovremo abituare. Infatti per ogni periferica integrata nel PIC vi sono una serie di registri per poterla gestire, e gli interrupt non fanno eccezione. Anche i configuration bits si possono configurare manipolando i registri ad essi associati (che sono memory mapped, ovvero mappati in memoria, cioè accessibili come normali locazioni di memoria), tuttavia il compilatore ci mette a disposizione delle istruzioni facili da adoperare per cui solitamente si usano quelle. Ritornando a noi, i registri che bisogna utilizzare per controllare gli interrupt sono riportati nel capitolo 9 del datasheet del PIC18F4550. Ce ne sono diversi e consentono di gestire gli interrupt associati a tutte le periferiche del PIC. Noi al momento avremo a che fare solo con alcuni bit dei registri INTCON e RCON.
I PIC18 come abbiamo detto possiedono due livelli di interruzioni, tuttavia possono lavorare anche con un solo livello, come avviene nei PIC16. Noi al momento useremo questa modalità, che quindi usa solo le interruzioni ad alta priorità. Per fare ciò dobbiamo settare a 0 il bit IPEN del registro RCON con il comando RCONbits.IPEN = 0. A questo punto dobbiamo abilitare le interruzioni globali settando il bit GIE del registro INTCON e le interruzioni delle periferiche attraverso il bit PEIE sempre del registro INTCON, ponendoli entrambi a 1.
In generale ogni periferica possiede un bit di abilitazione all'interno dei registri dedicati agli interrupt; inoltre vi è un altro bit che fa da flag, ovvero va a 1 quando si verifica l'interruzione. In particolare il nostro PIC presenta una linea di interruzione collegata alle PORTB, per cui una volta abilitata, con il bit RBIE di INTCON, se il livello logico sulle porte cambia il flag relativo alle PORTB, RBIF di INTCON, si porta a 1.
Abbiamo detto che quando si verifica un'interruzione il programma principale viene interrotto e

viene eseguita una Interrupt Service Routine, nella quale è specificato il codice da eseguire quando si verifica l'interrupt. Avendo due vettori di interruzione ovviamente sarà necessario scrivere due ISR: una verrà caricata quando si avrà un'interruzione ad alta priorità e l'altra quando si avrà un'interruzione a bassa priorità. Poiché la funzione main, prima di passare il controllo alla ISR, deve salvare il contesto, non possiamo usare una funzione normale per codificare una ISR, ma dobbiamo usare il qualificatore interrupt. L'ISR associata alle interruzioni ad alta priorità si scrive nel modo seguente:

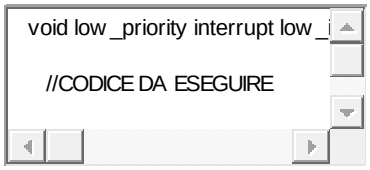
C




```
1 void interrupt high_isr(void){
2
3   //CODICE DA ESEGUIRE
4
5 }
```

Notiamo che la funzione non prende e non restituisce argomenti. Per l'ISR a bassa priorità dobbiamo aggiungere la parola chiave low_priority:

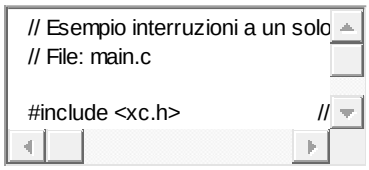
C




```
1 void low_priority interrupt low_isr(void){
2
3   //CODICE DA ESEGUIRE
4
5 }
```

Ovviamente tali funzioni non dovranno essere richiamate all'interno del main, ci penserà automaticamente il PIC quando si verificherà un'interruzione. Riportiamo dunque il codice completo del programma che fa lampeggiare il led su RB2 e permette di accendere il led su RB3 quando si preme il tasto su RB4:

C




```
1   // Esempio interruzioni a un solo livello di priorità
2   // File: main.c
3
4   #include <xc.h>                 //Header file generico
5
6   #define _XTAL_FREQ 48000000         //Specifico la frequenza dell'oscillatore
7
8   //Prototipi di funzione
9   void __delay_1ms(unsigned int count);
10 void interrupt high_isr(void);
11 void low_priority interrupt low_isr(void);
12
13 void main(void) {
14
15    TRISA = 0xFF;                //Imposto tutti i pin di PORTA come input
16    TRISB = 0xFF;                //Imposto tutti i pin di PORTB come input
17    TRISC = 0xFF;                //Imposto tutti i pin di PORTC come input
18    TRISD = 0xFF;                //Imposto tutti i pin di PORTD come input
19    TRISE = 0xFF;                //Imposto tutti i pin di PORTE come input
20
21    TRISBbits.TRISB2 = 0;          //Imposto il pin RB2 come output
22    TRISBbits.TRISB3 = 0;          //Imposto il pin RB3 come output
23
24    RCONbits.IPEN   = 0;           //Disabilito livelli di priorità
```

```
25    INTCONbits.RBIE = 1;            //Abilito interruzioni su PORTB
26    INTCONbits.GIE  = 1;            //Abilito interruzioni globali
27    INTCONbits.PEIE = 1;            //Abilito interruzioni sulle periferiche
28
29    while (1) {                  //Ciclo infinito
30
31       PORTBbits.RB2 = 1;          //Led acceso
32       __delay_1ms(250);           //Ritardo di 250 ms
33       PORTBbits.RB2 = 0;          //Led spento
34       __delay_1ms(250);           //Ritardo di 250 ms
35
36    }
37
38 }
39
40 void __delay_1ms(unsigned int count){
41
42    while(count-- != 0){
43
44       __delay_us(1000);           //Ritardo di 1 ms
45
46    }
47
48 }
49
50 //ISR per interrupt ad alta priorità
51 void interrupt high_isr(void){
```

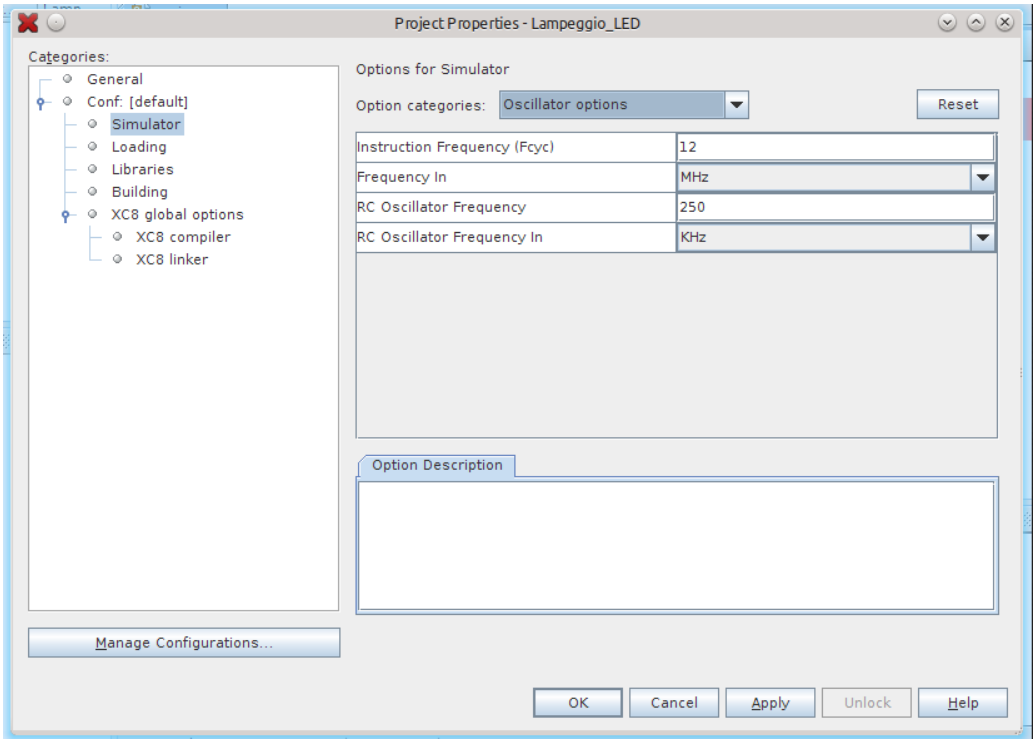
```
52
53    if (INTCONbits.RBIF == 1){       //Verifico che PORTB abbia generato
54                                 //l'interruzione
55
56        __delay_1ms(100);           //Delay di 100 ms per debounce
57
58        if (PORTBbits.RB4 == 0)      //Verifico che il tasto premuto sia RB4
59            PORTBbits.RB3 = 1;       //Accendo RB3
60        else
61            PORTBbits.RB3 = 0;       //Spengo RB3
62
63        INTCONbits.RBIF = 0;         //Resetto il flag d'interruzione su PORTB
64
65    }
66
67 }
68
69 //ISR per interrupt a bassa priorità
70 void low_priority interrupt low_isr(void){
71
72 }
```

Quanto fatto nel main è già stato spiegato. Notiamo solo che il bit di abilitazione degli interrupt su PORTB viene settato prima degli interrupt globali. Per quanto riguarda la ISR associata agli interrupt a bassa priorità è vuota, perché essi sono disabilitati (avremmo anche potuto ometterla), mentre nell'ISR ad alta priorità si effettua prima un controllo sul flag relativo a PORTB, perché l'interruzione potrebbe essere stata generata da qualche altra periferica (infatti l'ISR è sempre la stessa), e poi si agisce di conseguenza, ovvero si effettua un controllo sul pin di nostro interesse e a seconda del suo stato si accende o si spegne il led. Infine si va a resettare il flag d'interruzione su PORTB. Notiamo la presenza di un delay di debounce (antirimbalzo), che serve a filtrare eventuali rimbalzi che porterebbero a commutare il segnale sul pin più di una volta. Infatti, essendo il tasto un dispositivo meccanico, quando esso viene premuto il passaggio 0 -> 1 non è pulito, ma vi possono essere delle transizioni spurie a causa del rumore sovrapposto.

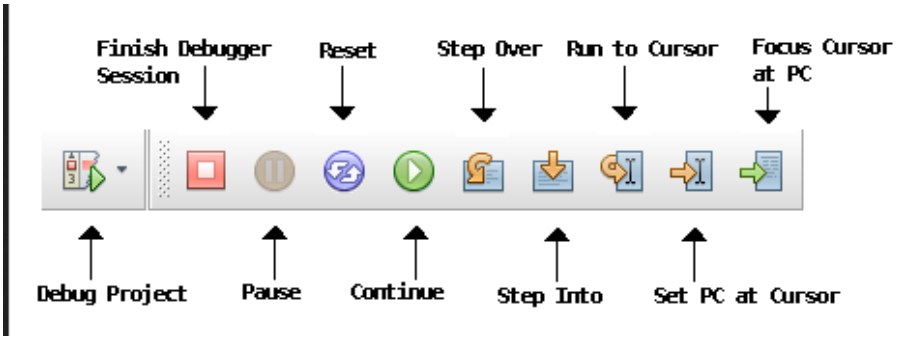
In alternativa all'istruzione INTCONbits.GIE = 1 necessaria per abilitare gli interrupt globali, l'XC8 mette a disposizione la funzione ei(), che in pratica fa la stessa cosa. Abbiamo anche una funzione di() che serve a disabilitare le interruzioni globali, ovvero a porre INTCONbits.GIE = 0. Visto che di norma è necessario settare anche altri bit, tali funzioni le trovo inutili, quindi tanto vale usare direttamente i registri.
Nel caso in cui avessimo voluto usare entrambi i livelli di interruzione, avremmo dovuto porre a 1 il bit IPEN. In tal caso i bit GIE e PEIE assumono un significato differente e si rende necessario associare una priorità ad ogni periferica utilizzata. In ogni caso quando ne avremo bisogno, mostrerò come procedere.
Per maggiori approfondimenti sull'argomento consiglio di consultare il capitolo 9 del datasheet del nostro PIC e il paragrafo 5.9 del manuale del compilatore XC8

Il simulatore di MPLAB X

Una volta che abbiamo scritto il programma, non è detto che esso si comporti come ci aspettiamo, oppure che sia privo di errori. In generale, il ciclo di sviluppo di un software comprende anche una fase di debugging, necessaria per individuare e risolvere eventuali errori o malfunzionamenti che possono presentarsi durante l'esecuzione dello stesso. Ci si affida solitamente ad un debugger, che può essere software oppure hardware. Un debugger hardware in pratica è un dispositivo, spesso integrato nel programmatore, che agisce direttamente sul microcontrollore durante l'esecuzione del codice ed è capace, interfacciandosi con l'ambiente di sviluppo, di fermare l'esecuzione in punti precisi del codice, ispezionare registri, e tante altre utili funzionalità che consentono di individuare facilmente eventuali problemi. Un debugger software invece non è nient'altro che un simulatore software implementato nell'ambiente di sviluppo, che appunto simula il funzionamento del microcontrollore e permette di fare le stesse cose che fa il debugger hardware. Ovviamente il debugger software ha dei limiti rispetto a quello hardware, che consistono nella velocità di esecuzione del codice, nel numero e tipo di periferiche che può emulare, e così via. Tuttavia risulta particolarmente utile per i nostri scopi. Voglio aggiungere che il simulatore per il PIC18F4550 è ancora in fase di sviluppo, per cui alcune funzionalità che sono disponibili in MPLAB 8 non sono ancora state implementate in MPLAB X.
Per utilizzare il simulatore, la prima cosa che dobbiamo fare è impostare la corretta frequenza di funzionamento del microcontrollore. Apriamo perciò le proprietà del progetto cliccando su File -> Project Properties, quindi visualizziamo la scheda Simulator tra le varie categorie disponibili, infine dal menù a tendina scegliamo Oscillator options (figura 1). Inseriamo ora il valore della FCYC della nostra configurazione, ovvero 12 Mhz, nel campo Instruction Frequency (Fcyc) e clicchiamo su Ok.

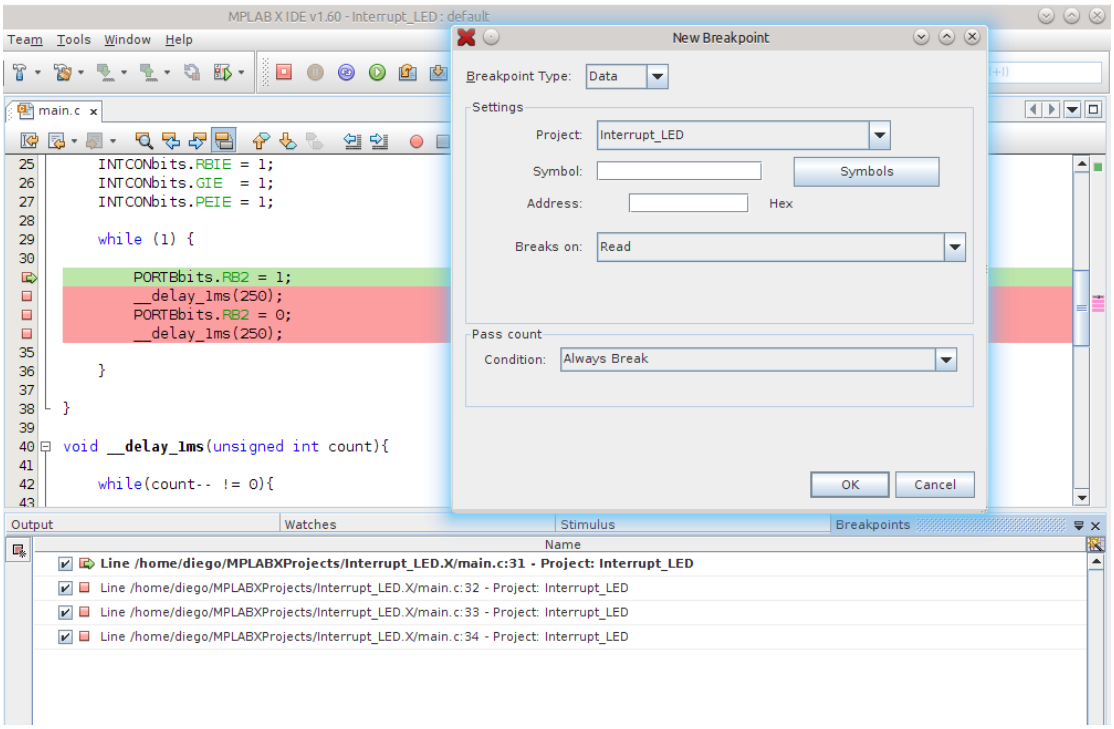


Per avviare la sessione di debug clicchiamo su Debug Project, accessibile dal menù Debug, oppure attraverso la toolbar in alto. Il programma verrà prima compilato, se ci sono modifiche rispetto all'ultima compilazione, dopodiché verrà eseguito in modalità di debug. L'output del debug verrà mostrato in basso, nella Debugger console. Una cosa che consiglio di fare è abilitare la toolbar per il debug (figura 3), che ci consente di controllare l'esecuzione del programma in maniera più agevole. Ciò si può fare mettendo la spunta su View -> Toolbars -> Debug. Dopo aver fatto partire l'esecuzione, la possiamo fermare attraverso il tasto Pause, riprenderla con Continue, resettare il programma con il tasto Reset ed interrompere la sessione di debug con Finish Debugger Session.

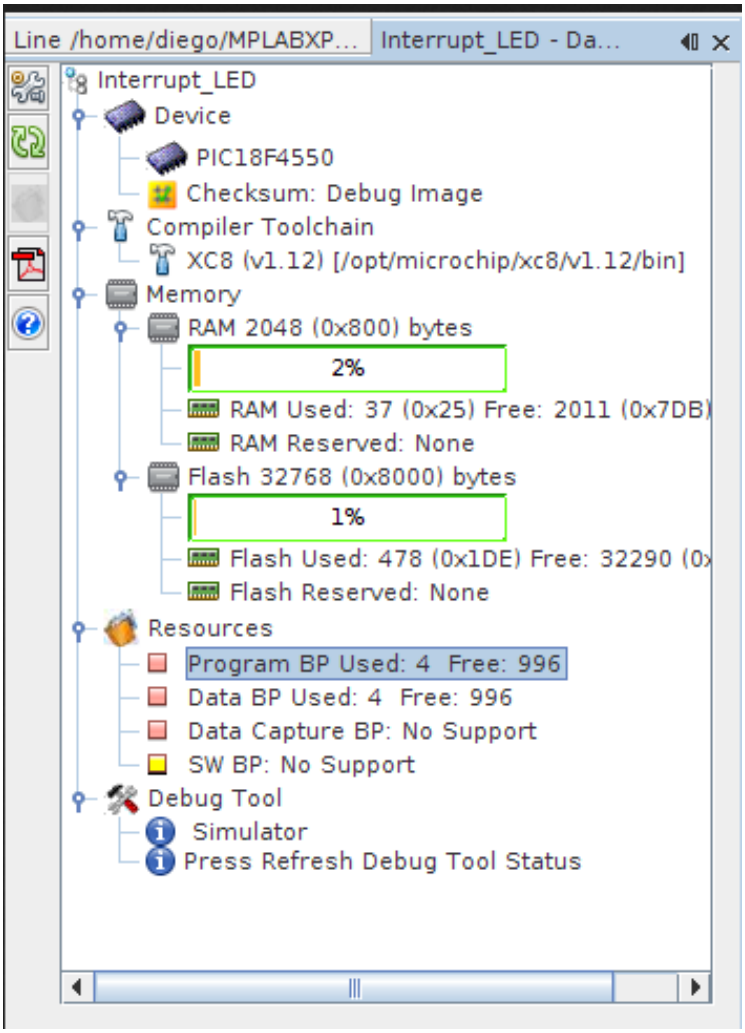


Nella fase di debugging è utile fermare l'esecuzione del codice in determinati punti, in modo che sia possibile eventualmente esaminare i valori contenuti nei registri del PIC, le variabili del codice, ecc. Per fare ciò ci serviamo dei Breakpoint, che servono a bloccare l'esecuzione nel punto in cui sono inseriti. Per inserire un breakpoint basta fare clic con il mouse sul margine sinistro dell'editor di testo, a fianco dell'istruzione in corrispondenza della quale si vuole arrestare l'esecuzione, mentre per eliminarlo basta rifare clic sul breakpoint. Possiamo vedere tutti i breakpoint che abbiamo inserito attraverso la finestra Breakpoints, che si apre cliccando su Window -> Debugging -> Breakpoints. In questa scheda si dà la possibilità di inserire ulteriori breakpoint cliccando con il tasto destro in un punto qualsiasi e aprendo la finestra New Breakpoints (figura 4), dove si possono impostare utili proprietà. Nella scheda Breakpoints è possibile anche disabilitare alcuni breakpoint

momentaneamente oppure eliminarli, modificarne le proprietà, ecc. Per maggiori informazioni fate riferimento alla guida online di MPLAB X, accessibile dall'ambiente di sviluppo attraverso Help -> Help Contents.

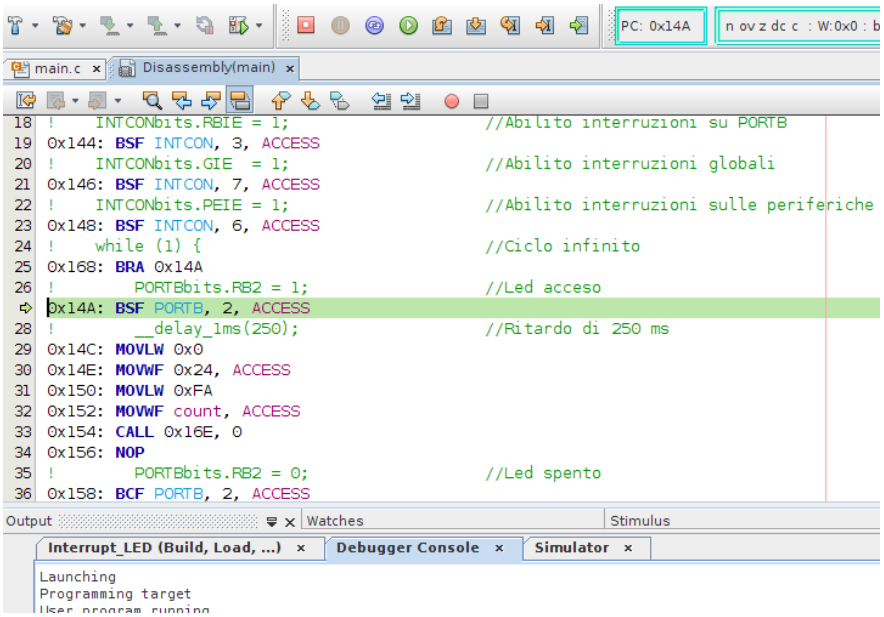


Solitamente i debugger hardware hanno un numero limitato di breakpoint che possono inserire, mentre nel simulatore non abbiamo questo limite. La dashboard ci mostra quanti breakpoint ci rimangono (figura 5).



Per muoverci all'interno del codice un passo alla volta si usano le funzionalità di Step, controllabili attraverso la toolbar che abbiamo abilitato. Abbiamo la possibilità di avanzare una riga di codice alla volta con la funzione Step Over. In questo caso se viene incontrata una chiamata a funzione, essa viene eseguita per intero e poi il programma viene arrestato. La funzione Step Into è simile, con la differenza che se vene incontrata una chiamata a funzione viene eseguita solo la prima riga di codice della funzione. Altra possibilità è quella di eseguire il programma fino al punto in cui è posizionato il cursore lampeggiante nell'editor (Run to Cursor), oppure posizionare il program counter nel punto in cui si trova il cursore (Set PC at Cursor). Infine c'è la possibilità di spostare il cursore nel punto in cui si trova il program counter (Focus Cursor ad PC). Altre funzionalità di step sono Step Out e Animate, che però nella versione corrente del simulatore per PIC18F4550 non sono implementate.

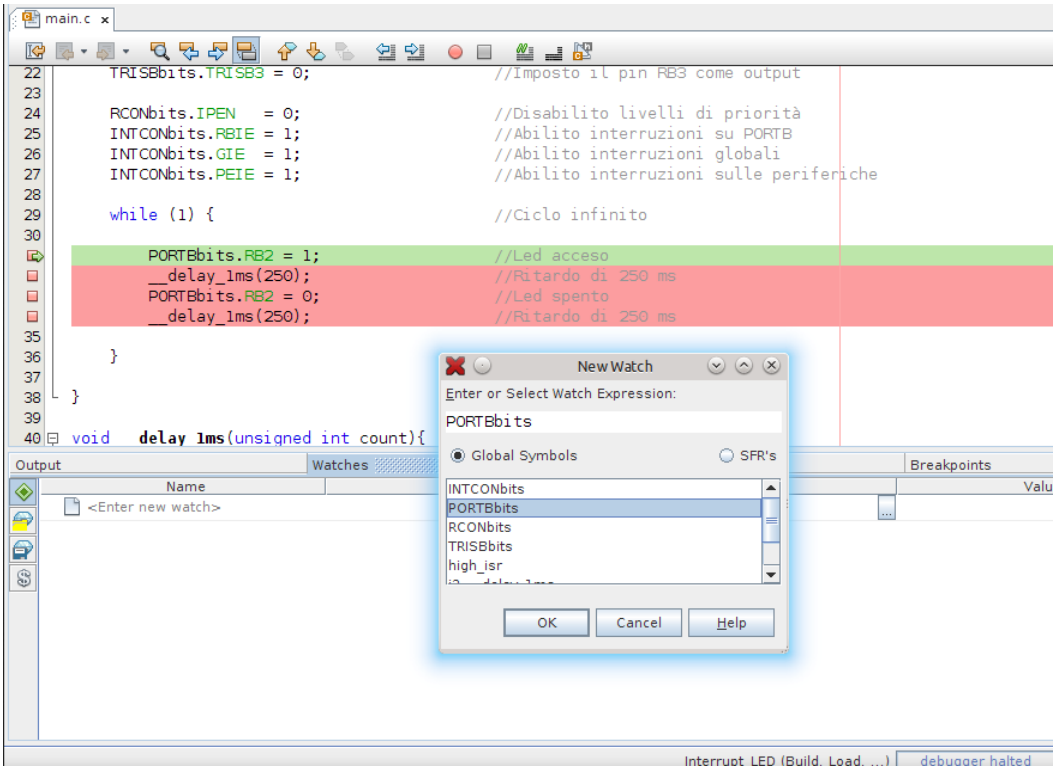
Un'interessante possibilità è quella di analizzare l'avanzare del programma attraverso il codice assembly generato dal compilatore, aprendo la finestra Window -> Debugging -> Disassembly (figura 6). Una versione statica del listato assembly è visualizzabile attraverso Window -> Output -> Disassembly Listing File, che poi si tratta del file listing.disasm presente sotto la directory disassembly del progetto.

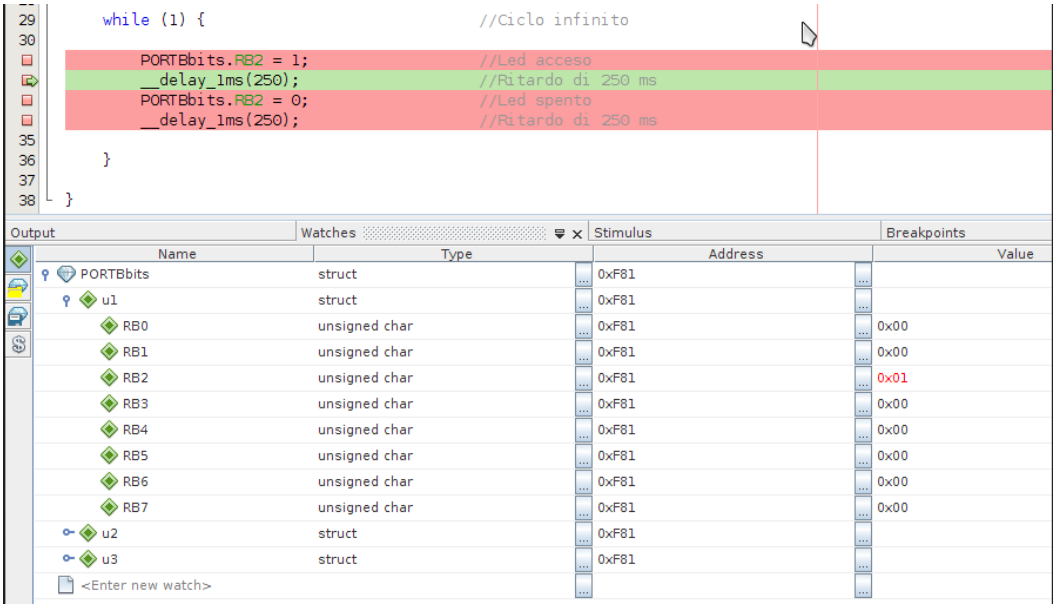


Abbiamo detto che uno dei motivi per cui si va ad arrestare l'esecuzione in punti precisi del programma è per controllare i valori di eventuali variabili o registri del microcontrollore, in modo da verificare se tali valori sono quelli attesi oppure no. Ciò lo possiamo fare mediante le schede Watches e Variables, accessibili sempre sotto il menù Window -> Debugging. La prima ci permette di visualizzare il valore di simboli globali presenti nel codice C oppure dei registri interni del PIC. Una volta aperta la scheda Watches, basta fare clic con il tasto destro in un punto qualsiasi della stessa, quindi cliccare su New Watch. Si aprirà una schermata, come mostrato nella figura 7, nella quale possiamo scegliere tra i simboli globali elencati, oppure, mettendo la spunta su SFR's, selezionare uno dei tanti registri interni del PIC. Facciamo riferimento all'ultimo programma scritto e mettiamo in corrispondenza di ogni istruzione presente nel ciclo while un breakpoint. Possiamo vedere come varia il valore del registro PORTB scegliendo nella finestra New Watch, tra le variabili globali, la PORTBbits. Essa comparirà nella scheda Watches, in basso; espandiamone in contenuto, in particolare la struct u1. Avviando la sessione di debug il nostro programma si arresterà in corrispondenza dell'istruzione

```
PORTBbits.RB2 = 1; //Led acceso
```

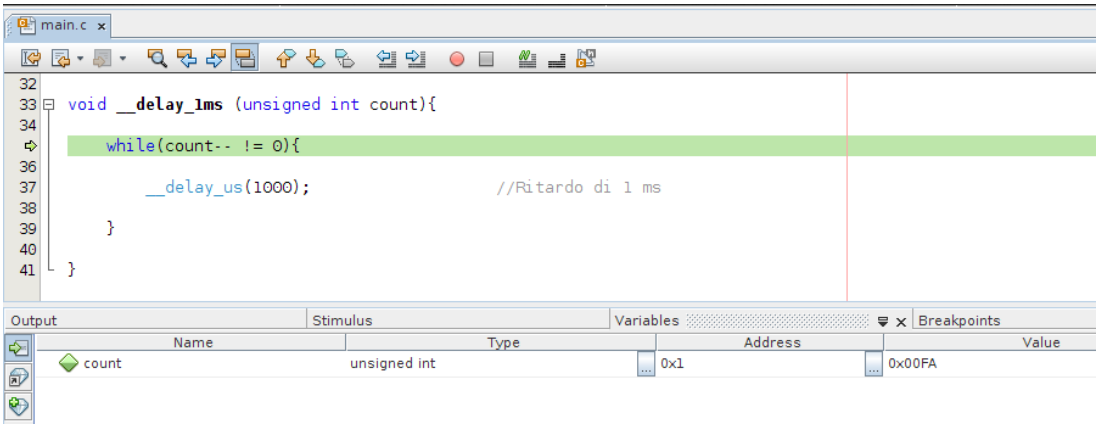
e noteremo come il valore di RB2 sia ancora pari a 0. Continuiamo l'esecuzione cliccando su Continue e vedremo che a questo punto il valore del bit diverrà pari ad 1 e di colore rosso (infatti le variabili che cambiano valore vengono evidenziati con questo colore), come mostrato in figura 8. Continuando ulteriormente ad eseguire il programma vedremo questo registro cambiare continuamente.



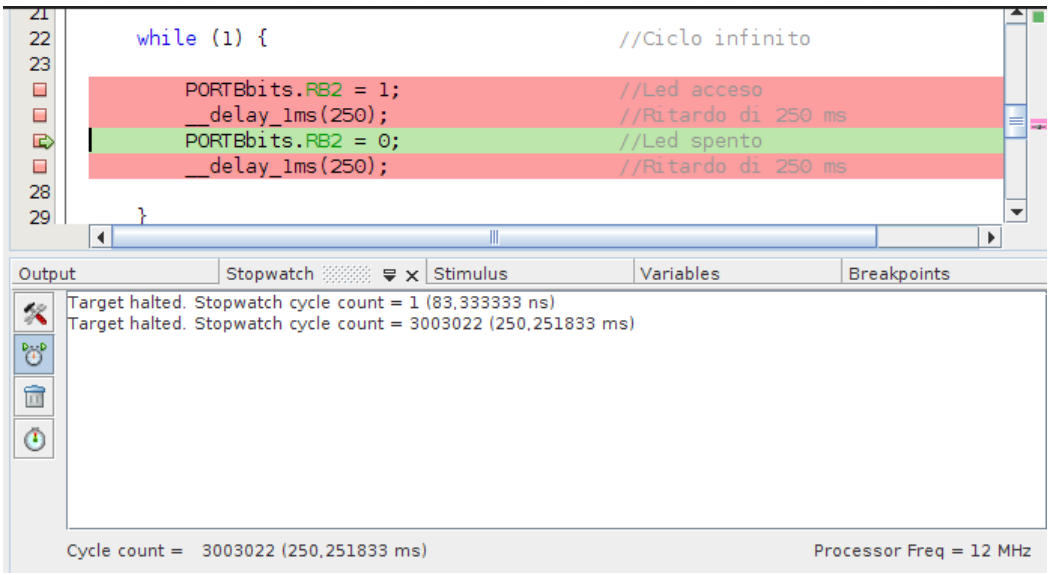


La scheda Variables invece ci permette di ispezionare il valore delle variabili locali del nostro programma. Diversamente da prima, non dobbiamo inserire noi le variabili da controllare, ma esse verranno aggiunte dinamicamente durante la fase di debug. Deselezionando il tasto sulla sinistra che recita “Show only variables used near PC location” visualizzeremo solo le variabili in prossimità del cursore. Nel nostro programma di lampeggio led non abbiamo dichiarato alcuna variabile nel main, per cui effettuando il debug la schermata Variables resterà vuota, mentre ne abbiamo una (count) nella funzione __delay_1ms. Per visualizzarla arriviamo al breakpoint posto in corrispondenza di uno dei due delay nel ciclo while e clicchiamo su Step Into, che ci farà entrare nella funzione e si bloccherà in corrispondenza della prima istruzione (non esegue la funzione per intero, come accade con Step Over). Se adesso andiamo a guardare nella schermata Variables vedremo comparire la variabile count (figura 9), il cui valore iniziale è proprio pari 0xFA (250 in decimale). Adesso possiamo spostarci all’interno di questa funzione una istruzione alla volta con

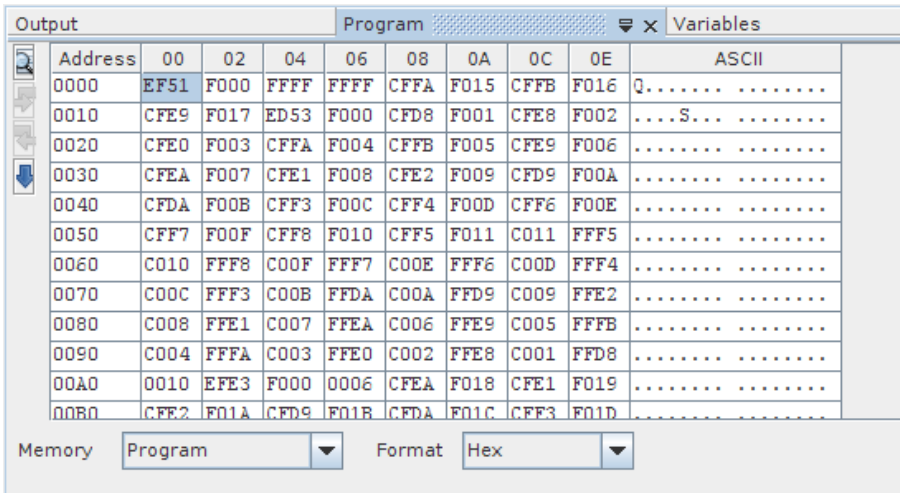
Step Over, e vedremo appunto la variabile count decrementarsi di volta in volta di un'unità. Una cosa interessante che è possibile fare è modificare il valore delle variabili (o dei registri) a run time, ovvero a tempo di esecuzione, durante la fase di debug. Proviamo a cambiare il valore della variabile count, ad esempio mettendola a 5, e vedremo che essa inizierà a decrementarsi a partire da questo nuovo valore.



Un altro tool interessante è lo Stopwatch, accessibile sempre sotto Window -> Debugging, che permette di misurare il tempo che intercorre tra un breakpoint e un altro. Avviamo lo Stopwatch e facciamo eseguire il programma, vedremo che all'interno della scheda compariranno le misure temporali tra un breakpoint e il successivo (figura 10). Possiamo ad esempio verificare che il ritardo tra l'accensione e lo spegnimento del led sia proprio pari a 250 ms. Attraverso i controllo laterali è possibile resettare lo stopwatch e pulire la storia dei ritardi.

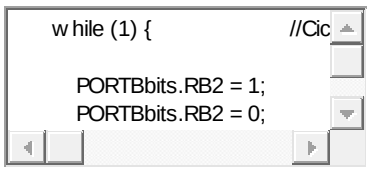


In MPLAB X è molto facile visualizzare il contenuto di tutte le memorie del PIC, in particolare la memoria flash, la ram, la EEPROM, i registri e così via. Ciò è possibile grazie alle schede accessibili attraverso Window -> PIC Memory Views. Possiamo facilmente cambiare il tipo di memoria e il formato di visualizzazione dei dati mediante i menu a tendina in basso (figura 11). Le locazioni di memoria inoltre vengono aggiornate dinamicamente durante il debugging.



Il simulatore ci mette a disposizione un altro strumento utile che è il Logic Analyzer, accessibile attraverso Window -> Simulator -> Analyzer, che consente di visualizzare in maniera grafica le variazioni dei segnali digitali, proprio come un analizzatore di stati logici. Modifichiamo il ciclo while del programma come di seguito:

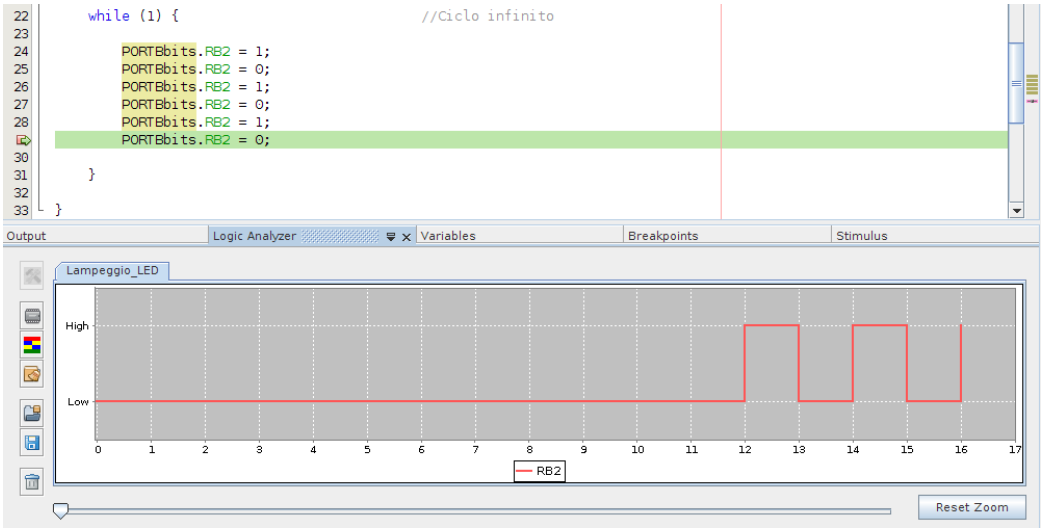
C




```
1     while (1) {                 //Ciclo infinito
2
3         PORTBbits.RB2 = 1;
4         PORTBbits.RB2 = 0;
5         PORTBbits.RB2 = 1;
6         PORTBbits.RB2 = 0;
7         PORTBbits.RB2 = 1;
8         PORTBbits.RB2 = 0;
9
10    }
```

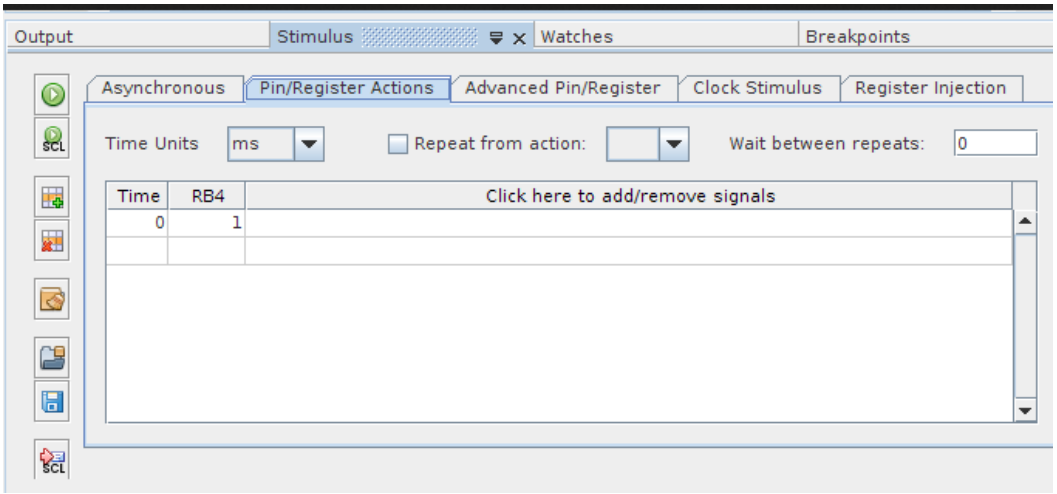
Inseriamo un breakpoint in corrispondenza dell'ultima istruzione del ciclo ed avviamo il Logic Analyzer. Per aggiungere la visualizzazione di una traccia (che può essere un pin o un bus) clicchiamo su Edit pin channel definitions. Tra i pin del microcontrollore scegliamo RB2, aggiungiamolo nell'elenco a destra e clicchiamo Ok. Ora avviando la sessione di debug comparirà nella scheda del Logic Analyzer il grafico del segnale su RB2, che commuta alternativamente tra 0

e 1, come mostrato nella figura 12. Possiamo fare in modo che la traccia sia perfettamente centrata a schermo cliccando con il tasto destro nella finestra, dunque Dimensiona automaticamente -> Asse Orizzontale. Possiamo poi spostarci orizzontalmente con il cursore in basso, mentre è possibile resettare la visualizzazione con il tasto Reset Zoom. Per eliminare tutti i segnali visualizzati si può cliccare sul tasto Clear channel definitions, posto sulla sinistra della scheda.

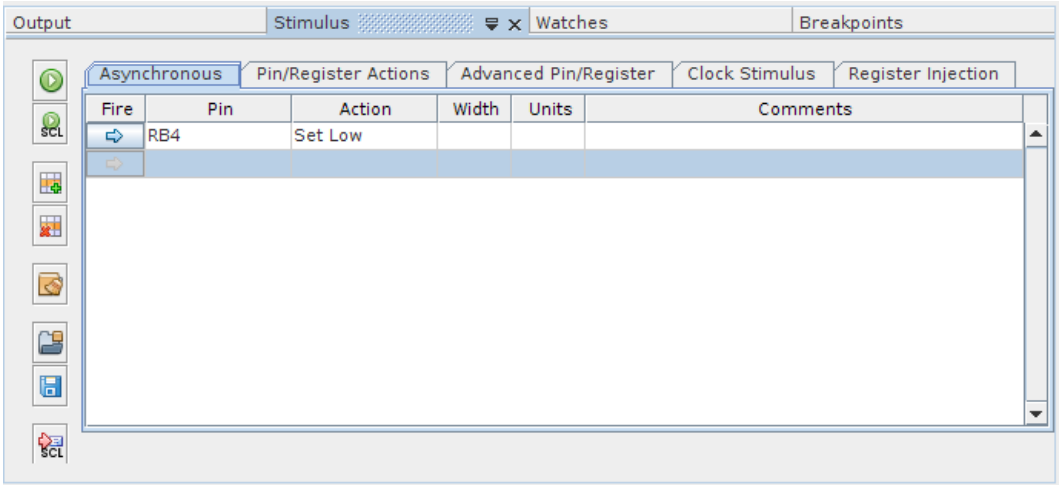


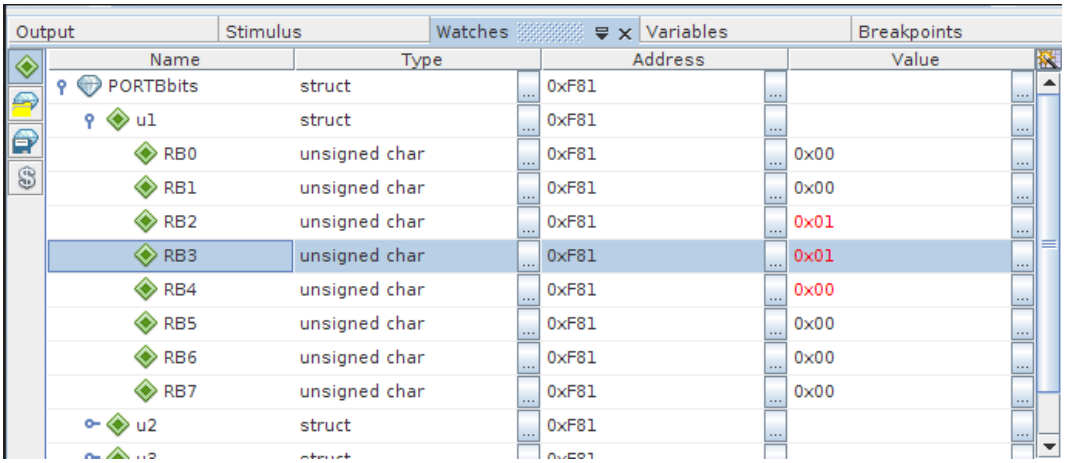
Durante la simulazione può capitare la necessità di ricevere stimoli dall'esterno. Uno stimolo è in pratica la simulazione di un evento hardware, come ad esempio la pressione di un tasto, oppure la scrittura di dati in memoria o in un registro del PIC. Si possono avere due tipi di stimoli: asincroni, che avvengono una sola volta e sono azionati dall'utente, e sincroni, che sono invece degli eventi pianificati in maniera opportuna. Per definire uno stimolo possiamo usare la finestra Window -> Simulator -> Stimulus, la quale a sua volta conterrà diverse schede.
Nella scheda Asynchronus è possibile settare gli stimoli asincroni, uno su ogni riga della tabella. Il pin del microcontrollore sul quale agire va selezionato sotto la colonna Pin, mentre l'azione da compiere va specificata sotto la colonna Action: possiamo porre il pin al valore logico alto (Set High) o basso (Set Low), farlo commutare (Toggle) e applicare un impulso alto (Pulse High) o basso (Pulse Low). Nel caso di impulsi possiamo anche specificare la durata temporale sotto la colonna Width, mentre sotto la colonna Units si va a specificare l'unità di misura della durata. Infine possiamo aggiungere un commento (opzionale) sotto la colonna Comments. Lo stimolo potrà essere applicato in qualsiasi istante durante esecuzione in modalità debug con la pressione della corrispondente freccia, posta sotto la colonna Fire.
Nella scheda Pin/Register Actions si possono invece impostare gli stimoli sincroni. Anche in questo caso si procede riga per riga, in ognuna delle quali si specifica l'instante temporale (assoluto) in cui applicare lo stimolo e i valori logici che i vari pin devono assumere. Per aggiungere uno o più registri da "stimolare" basta fare clic sul titolo della colonna che riporta Click here to add/remove signals. A questo punto i registri da stimolare compariranno sulle colonne della finestra. In Time andremo ad inserire le scadenze temporali, ad esempio se vogliamo che il pin RB4 sia a 1 per i primi 100 ms di esecuzione del programma andremo a posizionare due righe, la prima con il parametro Time impostato a 0 e valore logico di RB4 a 1, e una seconda riga in cui Time è posto a 100 e RB4 a 0. Ovviamente bisognerà scegliere l'unità di misura corretta del tempo sotto il menù a tendina Time Units. Selezionando il checkbox Repeat from action e specificando il valore dal menu a tendina possiamo ripetere lo stimolo a partire dall'ultimo eseguito. Si può specificare anche un delay tra le varie ripetizioni.
È possibile configurare gli stimoli sincroni in maniera più sofisticata attraverso la scheda Advanced Pin/Register, che non vediamo, così come non approfondiamo la scheda Clock Stimulus, nella quale è possibile applicare un segnale temporizzato sui pin del microcontrollore e la scheda Register

Injection, che permette di inserire dei valori nei registri.
Vediamo adesso un caso pratico di applicazione di stimoli al nostro programma. Abbiamo visto che accanto al programma principale, che fa lampeggiare un led, ovvero pone alternativamente a 0 e a 1 l'uscita RB2 del PIC, c'è la possibilità di accendere il led su RB3 qualora si vada a premere il tasto su RB4. Ovviamente nel simulatore non abbiamo l'hardware di contorno del PIC, per cui possiamo pensare di simulare la pressione del tasto con gli stimoli.
Il tasto, come abbiamo visto è attivo basso, quindi dovremo simulare prima di tutto la resistenza di pull-up, che pone il pin RB4 al valore logico alto. Ciò lo possiamo fare nella scheda di settaggio degli stimoli sincroni. Aggiungiamo il pin RB4 e poniamo il suo valore a 1, mentre in Time mettiamo 0 (scegliamo un'unità di misura di tipo temporale, ad esempio ms), come mostrato in figura 13. Ora premiamo sulla sinistra il tasto con la freccia verde (Apply synchronus stimulus) per applicare lo stimolo, che partirà in sincronia con l'esecuzione del programma. Adesso apriamo la scheda Watches, scegliamo di visualizzare il registro PORTBbits e avviamo la sessione di debug; potremo notare che RB4 è sempre pari ad 1.



Per simulare la pressione del tasto, che in pratica porta a 0 il valore logico su RB4 quando premuto, utilizziamo uno stimolo asincrono. Nella tabella Asynchronus scegliamo RB4 come pin e Set Low come azione, come mostrato nella figura 14. Ora, mentre gira la sessione di debug, clicchiamo sulla freccia posta sotto Fire e mettiamo in pausa il programma. Potremo osservare come RB4 si sia portato a 0 e RB3 a 1, che è il comportamento che ci aspettiamo (figura 15).





In questo articolo abbiamo analizzato le funzionalità di base del simulatore, che sono sufficienti per i nostri scopi. Tuttavia le potenzialità di questo strumento non si limitano a quanto visto fino ad ora. Vi sono altre funzionalità più avanzate come ad esempio il Simulator Trace, che permette di effettuare una registrazione passo passo del codice eseguito, la simulazione delle periferiche USART, dell'interfaccia di memoria esterna e tanto altro. Come ho già detto in precedenza, si può consultare la guida online dell'ambiente di sviluppo, in particolare la sezione MPLAB X Simulator, per approfondirne l'uso e scoprire ulteriori possibilità.
Per questa lezione è tutto, nella prossima finalmente mostreremo come utilizzare il bootloader della Microchip, in modo da caricare direttamente il codice via USB invece di riprogrammare il PIC di volta in volta

PIC & USB: programmazione via bootloader

Il bootloader è un programma che, caricato nel microcontrollore, permette di programmare il microcontrollore stesso direttamente tramite la porta seriale o USB, invece di dover utilizzare un programmatore. Questo permette di velocizzare notevolmente il processo di sviluppo del firmware. Nella trattazione seguente si farà riferimento al bootloader via USB per PIC della serie 18FXXXX con porta USB, avendo preventivamente installato il software necessario.

Per poter utilizzare questa possibilità è necessario programmare il PIC con il codice riportato nel file `USB Device - HID - HID Bootloader - C18 - PIC18F4550.hex` contenuto nella cartella `Files_per_Bootloader_pic18F2550`. Anche se è stato sviluppato per PIC18F4550, è perfettamente compatibile con gli altri tre PIC della serie 18FXXXX dotati di porta USB.

Per la programmazione utilizziamo il PicKit3 con il programma MPLAB IPE. Dovremo selezionare il micro PIC18F2550 e dopo aver abilitato la modalità avanzata attivare l'alimentazione dal menù Power. Caricheremo il file hex citato sopra e dopo aver cancellato il micro lo programmeremo.

Ora si può collegare il circuito alla porta USB del PC e far entrare il sistema nella modalità bootloader semplicemente premendo e mantenendo premuto il pulsante denominato "RB4", premendo e rilasciando il pulsante "Reset" e quindi rilasciando il pulsante "RB4".

La prima volta che si entra in questa modalità, il PC riconosce una nuova periferica hardware, quindi inizia la procedura di identificazione della periferica e di ricerca dei relativi driver.

Il software per PC

Ora che il nostro PIC è stato programmato con il bootloader fresco di giornata, non ci resta altro che interfacciarlo con il PC mediante un software dedicato. Il software in questione ci viene fornito sempre dalla Microchip nelle application libraries e si trova sotto la cartella utilities prima menzionata. Gli utenti Windows in questo caso sono più fortunati ed hanno già i binari pronti per essere utilizzati, mentre gli utenti Linux dovranno compilarsi il software da sorgente. Tuttavia poiché anch'io sono un utente Linux e non mi piacciono le discriminazioni l'ho fatto per voi, quindi ecco qui sia i binari per Windows che per Linux.

HID bootloader software per PC (Linux)
HID bootloader software per PC (Windows)

Per dovere di cronaca, il software è stato compilato sotto Ubuntu 14.04 con QTCreator. Per fare ciò ho dovuto installare le librerie di sviluppo di libusb:

```
sudo apt-get install libusb-1.0-0-dev
```

In Linux, prima di avviare il programma dobbiamo dare i permessi di lettura e scrittura al file che rappresenta il nostro dispositivo USB (che può essere letto e scritto solo dall'utente root). Facciamo questo aggiungendo una regola al gestore dei dispositivi udev. Apriamo il terminale e diamo il seguente comando:

```
sudo echo -e '# Microchip HID Bootloader\nSUBSYSTEM=="usb",
ATTR{idVendor}=="04d8", ATTR{idProduct}=="003c", MODE="0666",
OWNER="nome_utente"' > /etc/udev/rules.d/99-hidbootloader.rules
```

dove al posto di nome_utente dovete inserire il vostro nome utente.
A questo punto avviamo l'applicativo con doppio clic su di esso (se avviato da linea di comando ci darà sulla console qualche info in più), che si presenterà come mostrato in figura 5.

Notiamo dalla figura come il software ci informi che non è collegato alcun dispositivo, questo perché esso non è in fase di programmazione. Per portare il bootloader in fase di programmazione dobbiamo tenere premuto il tasto su RB4 durante il reset, ovvero prima premiamo il pulsante su RB4 e lo lasciamo premuto, dopodiché premiamo e rilasciamo il pulsante di reset, infine rilasciamo

RB4. Se tutto è stato eseguito correttamente, il software mostrerà qualcosa simile a quanto riportato in figura 6, ad indicare che il dispositivo è stato riconosciuto. A questo punto possiamo aprire il file .hex dell'application code andando su File->Import Firmware Image o cliccando sulla corrispondente icona e flasharlo con Program->Erase/Program/Verify Device o ancora una volta cliccando sulla corrispondente icona. La procedura consiste prima nel cancellare la memoria del microcontrollore, poi nel programmarla ed infine nel confrontare i dati scritti con quelli letti. In figura 7 è mostrato l'esito di una programmazione.

Il software permette anche di resettare il dispositivo mediante l'apposito tasto nella toolbar, oppure attraverso Program->Reset Device, e di impostare in una finestra a parte quale memoria del microcontrollore scrivere, a cui si accede con Program->Settings… (figura 8). Attenzione a non mettere la spunta su Config Bits.
Ora se siete stati frettolosi ed avete programmato il PIC con i vecchi file .hex che usavate con il programmatore avrete notato come essi non funzionano. Infatti dobbiamo fare un passo indietro e predisporre i nostri programmi per il funzionamento con bootloader.

È il turno dell'application code

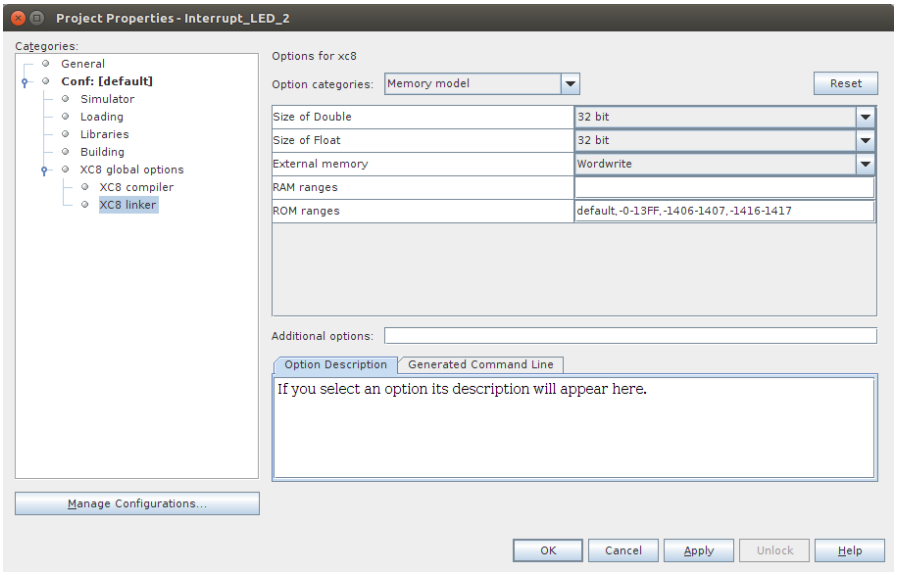
Nelle applicazioni che andrete a scrivere dovete tener conto di alcune limitazioni imposte dall'utilizzo del USB HID bootloader.

- Usare gli entry points rilocati per RESET e interrupt,
- Non usare il piedino RB4 (utilizzato nel USB HID bootloader),
- Non utilizzare il piedino D0 con il PIC18F4550 (viene utilizzato per segnalare con un LED l'attività del USB HID bootloader).

Abbiamo visto che al codice del bootloader è stata riservata una porzione di memoria programma che va dall'indirizzo 0 all'indirizzo 0x13FF. Questa parte di memoria non può più essere utilizzata dal programma utente, per cui dobbiamo rimuoverla dall'intervallo di default. Prendiamo uno dei nostri vecchi progetti, ad esempio quello sugli interrupt dello scorso articolo, e settiamo il range della ROM attraverso File->Project Properties, quindi XC8 global options-> XC8 linker, selezionando la voce Memory model nel menù Option categories in alto, come mostrato in figura 9, ovvero

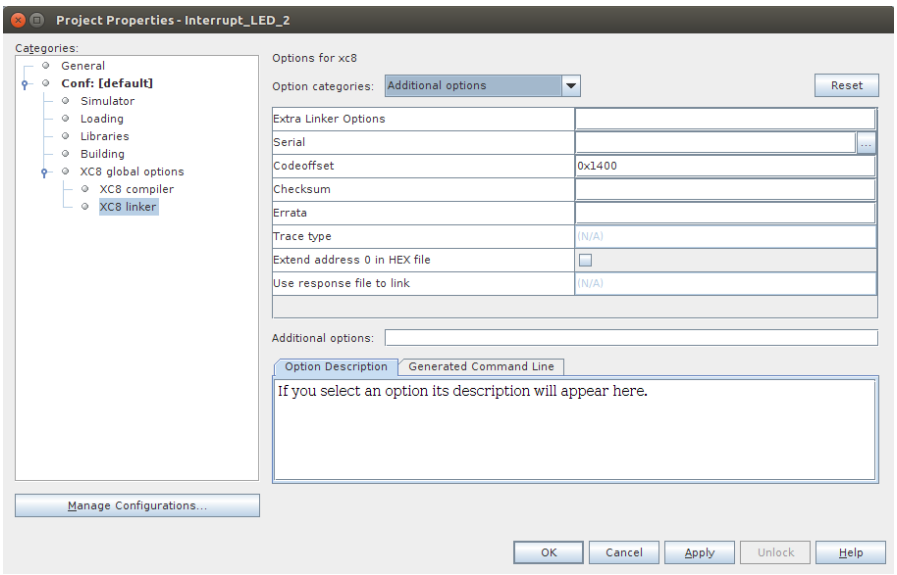
```
default,-0-13FF,-1406-1407,-1416-1417
```

In questo modo informiamo il linker che deve utilizzare l'intervallo di memoria di default privato della parte usata dal bootloader e delle locazioni 0x1406-0x1407 e 0x1416-0x1417, che sono usate per la firma di verifica e per la versione del bootloader.



Quanto fatto però non basta. Gli indirizzi del reset vector e degli interrupt vector sono fissi, dunque il compilatore continuerebbe ad ancorarli agli indirizzi di default, che però sono ora gestiti dal bootloader. Esiste tuttavia un'altra opzione del compilatore pensata proprio per l'uso con bootloader, che permette di effettuare una "traslazione rigida" di tutto il codice all'interno della memoria di una quantità specificata, inclusi i vettori di reset e interrupt. Si tratta dell'opzione Codeoffset, che possiamo specificare nella schermata Additional options, sempre sotto le proprietà del linker, come mostrato in figura 10, che va settata a

```
0x1400
```

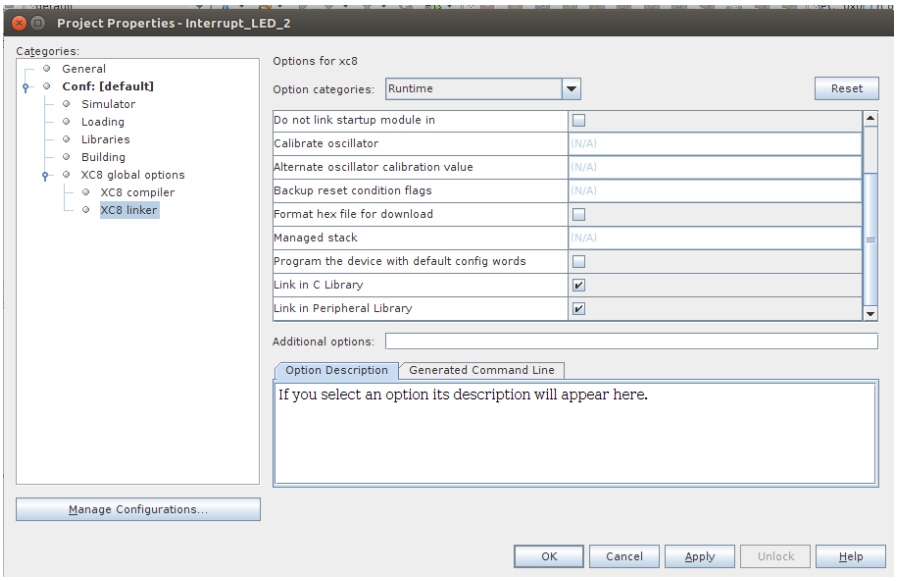


Quindi, se vogliamo utilizzare i nostri programmi in abbinamento al bootloader, dobbiamo specificare sempre in tutti i progetti queste due opzioni. Possiamo verificare che il compilatore abbia recepito correttamente le nostre direttive dando uno sguardo al file .map presente nella cartella

```
dist/default/production
```

del nostro progetto, ovviamente dopo averlo ricompilato. Qui possiamo vedere i parametri della linea di comando del linker. In linea di principio ci sono due tipi di opzioni passate al linker: quelle di tipo -p, che specificano dove collegare le locazioni di memoria fisse come il reset vector e gli interrupt vector, e quelle di tipo -A, che specificano gli intervalli in cui porre tutto il resto del codice. Notiamo come questi parametri specifichino i range di memoria che abbiamo impostato in MPLABX e che il reset vector sia ancorato adesso all'indirizzo 0x1400.

C'è un'ultima cosa da dire che riguarda i configuration bits. Dato che il bootloader è programmato nel nostro dispositivo per tutto il tempo, è buona norma che sia il bootloader ad impostarli. Infatti essi sono settati nel codice del bootloader, per cui nei nostri programmi non dovremo più specificarli, e anche se si potrebbe (infatti il software ha un'opzione per scriverli), non è consigliato, perché il nostro bootloader potrebbe non funzionare più correttamente. Ora però se non specifichiamo più i configuration bits nei nostri progetti, il compilatore XC8 tenderà ad impostare quelli di default. Quindi ultima cosa da fare è togliere la spunta alla voce Program the device with default config words, sotto la categoria Runtime del linker, come mostrato in figura 11.



Compiliamo quindi una volta per tutte il nostro codice applicazione e flashiamo il PIC con il file .hex ottenuto come visto nel paragrafo precedente. Ricordo che una volta effettuata la scrittura del programma il PIC deve essere resettato o con il tasto nella toolbar o con il pulsante fisico che abbiamo sulla nostra scheda.
Questo è tutto, ora possiamo tranquillamente fare a meno del programmatore e velocizzare la fase di testing dei nostri programmi.