
Esercitazioni C Enhanced Midrange

0C – Preparazione alle esercitazioni.

Questo corso è una serie di esempi per apprendere qualcosa su come impiegare il compilatore XC8 di Microchip nell'ambiente dei PIC, principalmente i recenti Enhanced Midrange.

Potrebbe intendersi come una continuazione di quanto dedicato all'Assembly, ma, nello stesso tempo, cerca di essere una trattazione non legata alla parte precedente.

Non occorre una speciale conoscenza del C, del quale si cerca di fornire alcune elementi di base, né una preparazione elettronica universitaria, ma senz'altro occorre avere idea di cosa si intende con programmazione e quali siano gli elementi fondamentali dell'elettricità (corrente, tensione, potenza, resistenza) e dell'elettronica (transistor, LED, logiche booleane, ecc.).

Questo perchè lavorare con i microcontroller non vuol dire scrive programmi del tutto estranei all'hardware, bensì avere a che fare direttamente con questo: la sola conoscenza del C non consente di utilizzare al meglio il microcontroller senza avere chiare le caratteristiche e le funzioni interne del componente. L'interdipendenza tra hardware e software è sempre fortemente presente.

Pertanto, dove necessario, c'è una trattazione abbastanza dettagliata dell'hardware su cui si sta lavorando: per quanto il C crei uno strato di astrazione tra hardware e software, senza sapere cosa fanno i moduli integrati del microcontroller diventa impossibile utilizzarlo.

A meno di voler fare uso esclusivo di librerie, che, però, quando mancano, impediscono completamente l'impiego della periferica, oltre a non fornire quanto potrebbe eservire per creare qualcosa autonomamente.

Una minima conoscenza delle istruzioni del linguaggio macchina sono utili, in quanto C permette di inserire nei sorgenti proprio degli elementi di Assembly, con lo scopo di ottimizzare le azioni sull'hardware.

Non trattiamo qui la storia e le origini del C. Ci basta sapere che C non è un linguaggio univoco, ma esiste in numerose versioni (dialetti) con caratteristiche anche sensibilmente diverse per essersi adeguate ad ambienti diversi.

In particolare, utilizzeremo il compilatore XC8 per le MPU a 8 bit di Microchip (che offre anche compilatori per i suoi prodotti a 16 e 32bit).

Il supporto è dato dall'ambiente di sviluppo MPLAB-X IDE.

Vediamo in queste prime pagine alcune nozioni che sono essenziali per poter procedere con sicurezza.



Nota: nel testo sono inseriti collegamenti ad altre pagine. Se le pagine sono in questo stesso testo, si accede direttamente. Se le pagine sono esterne, occorre un collegamento a internet. In tal caso è aggiunta l'indicazione (www).

Per anticipare alcune domande comuni, diciamo ora **cosa non è questo corso**:

- non è un corso teorico di C
- non è un corso teorico di XC8
- non è una serie di realizzazioni pratiche o commerciali.

Cosa è questo corso?

- è una serie di esempi di impiego del compilatore XC8 con i microcontroller a 8 bit di Microchip di ultima generazione
- è una serie di esempi di gestione delle varie funzioni integrate in questi microcontroller. Gli esempi hanno lo scopo di comprendere, almeno in linea di massima, come funzionano i moduli integrati e come gestirli con il C.
- cerca di dare risposte a molte domande che restano ignorate in altri ambiti.
- gli esempi hanno non hanno lo scopo di fornire realizzazioni copia-e-fai, ma di dare alcune idee di base per poter diventare autonomamente creativi e, almeno, capire meglio programmi scritti da altri.

A chi si rivolge?

- a chiunque è interessato, per qualsiasi ragione, all'impiego di PIC e C.

0.1 - Cosa serve per cominciare.

Quello che è necessario si può dividere in due parti:

- gli **strumenti di sviluppo** e
- l'**hardware di base**.

0.1.1 - Strumenti di sviluppo

Innanzitutto, è necessario disporre di **un PC con porte USB**.

Non occorre una macchina ad alte prestazioni: le compilazioni, per quanto complesse, sono processi che, con le CPU attuali, durano una manciata di secondi. Microchip consiglia CPU a partire da Pentium IV 2.6GHz con almeno 1GB di RAM o di un core i5 con 4GB per uso professionale. Da tenere presente che vecchie CPU, come ad esempio gli Athlon XP, possono non essere compatibili con le recenti versioni di Java.

Non occorre l'ultima versione del S.O. di Microsoft: Windows 7 va più che bene, come va bene 10. Il più vecchio XP è utilizzabile, ma le versioni di MPLAB-X successive alla 4.15 non sono supportate, come pure non sono supportati i più recenti tools come ICD4, PICKIT4 e SNAP e le applicazioni avanzate come MCC.

In effetti, però, non è neppure necessario Windows, in quanto l'ambiente di sviluppo di Microchip MPLAB-X è offerto anche per **MAC** o **Linux**.

Microchip consiglia s.o. a 64 bit per avere le migliori prestazioni, ma MPLAB-X è installabile anche su sistemi Windows a 32 bit; in tal caso, consiglia l'installazione di [JVM](http://www.java.com) (www). Per curiosità, gran parte di queste pagine sono scritte e sviluppate su una vecchia macchina XP con processore E6800, MPLAB-X 4.15 e PICKIT3.

Ovviamente serve **una connessione a Internet**, almeno per scaricare il software; in particolare, MPLAB-X è pensato per accedere direttamente a risorse del WEB, ma può funzionare anche stand alone.

La disponibilità di almeno un paio di **porte USB2.0 con capacità di corrente standard** è necessaria per collegare i tools di sviluppo, ma ormai tutti i personal computer ne dispongono. Da osservare, però, che **non sempre i port USB sono in grado di fornire la loro corrente nominale**, soprattutto nei casi di notebook e tablet. Qui potrà essere problematico alimentare l'hardware di sviluppo attraverso la connessione USB ed occorrerà un alimentatore esterno. Non servono port **USB3**, dato che, al momento, non ci sono tools di sviluppo con questa interfaccia.

Per testare **comunicazioni RS232 o 485** è utile disporre di una **porta seriale**. A questo riguardo va tenuto presente che è ben possibile che vi troviate in mano un PC privo di porte seriali e parallela, principalmente se utilizzate un portatile. Però esistono convertitori da USB a seriale o parallela.

Sarà anche utile un emulatore di terminale per verificare le comunicazioni seriali. Anche qui sono

disponibili numerose scelte, come [RealTerm](#), [Terminal.exe](#), [Hercules](#) (consigliato), ma anche il vecchio Hyperterminal di Windows.

La necessità di digitare **simboli** molto usati nel C (`{ } ~`), ma non presenti immediatamente sulla tastiera italiana, fa preferire al programmatore una **tastiera con layout USA**.

Inoltre, è molto utile avere un **buon monitor** ben definito in modo da non stancare la vista nella lettura dei listati e nelle sessioni di debug, dove l'attenzione è fissata a lungo su quanto appare sullo schermo. Un monitor di scarsa qualità affaticherebbe inutilmente la vista.

L'ambiente di sviluppo MPLAB-X e i compilatori XC occupano uno spazio limitato sul disco, così come i progetti relativi al lavoro eseguito, per cui è più che sufficiente avere 100GB liberi sull'hard disk da dedicare allo sviluppo dei microcontroller.

Sarà, però, opportuno avere una qualche strategia di backup, dato che perdere i file sorgenti dei lavori realizzati, anche non in ambiente professionale, può essere una situazione quanto mai spiacevole.

Sarà utile disporre anche di un **editor per programmazione** come [TextPad](#), [Notepad++](#) (consigliato), [Sublime Text](#), [SciTe](#), ecc. ([www](#)) o anche utilizzare l'editor dell'ambiente di sviluppo MPLAB-X.

Il banale Notepad di Windows andrebbe bene, ma l'uso di un editor specifico per programmazione offre l'indispensabile funzione di **colorazione del testo** che permette di distinguere immediatamente le varie funzioni di quanto scritto e risolve spesso errori e dimenticanze formali.

Per quanto riguarda gli strumenti software, sempre da **Microchip** possiamo scaricare gratuitamente sia l'ambiente di sviluppo **MPLAB-X**, sia il compilatore **XC8**.



Lo sviluppo dei controller di Microchip (PIC, AVR e SAM GCC) è basato sull'**ambiente integrato MPLAB-X** ([www](#)) IDE -Integrated Development Environment.

E' un programma espandibile che incorpora potenti strumenti per configurare, sviluppare, eseguire il debug e qualificare progetti per la maggior parte dei microcontroller di Microchip.

Questo sostituisce il vecchio MPLAB IDE, che è stato dismesso e non offre supporto ai componenti più recenti.

E' scaricabile gratuitamente dal [sito di Microchip](#) ([www](#))

Consente debug e simulazione ed è completato da varie funzioni, come l'ambiente MPLAB-IPE (Integrated Programming Environment) dedicato alla programmazione dei chip.



L'IDE supporta compilatori C per 8, 16 e 32 bit (XC8, XC16, XC32). A noi occorre la versione a 8 bit che dobbiamo [scaricare](#) ([www](#)) sempre dal sito di Microchip. Si tratta di un ANSI-C specificamente diretto alla programmazione dei microcontroller.

Il compilatore esiste in varie versioni legate alle caratteristiche delle varie famiglie a 8, 16 e 32 bit. **XC8** è per i processori a 8bit; di questa versione ne esistono tre tipologie, che si differenziano per la diversa ottimizzazione dell'eseguibile.

Anche se è possibile utilizzare dei **simulatori** per verificare il programma scritto, **non ne consigliamo l'uso**, in quanto le simulazioni offerte da programmi gratuiti o a basso costo non sono per niente utili a risolvere gli eventuali problemi che si devono affrontare e, in generale, sono solo impiego di tempo ben poco efficace.

Volendo usare una simulazione, l'ambiente MPLAB offre un **simulatore SIM** che permette di verificare quanto scritto senza l'impiego di alcun hardware.

Ben più efficace è il debug on-circuit sfruttando il motore ICD presente negli Enhanced Midrange (e in vari altri chip a 8 bit).

0.1.2 - Hardware di base

Se con **SIM** di MPLAB si può fare a meno (fino ad un certo punto, però) dell'hardware ed effettuare una simulazione, una verifica, un test, un debug, queste azioni si effettuano molto più efficacemente disponendo dell'hardware richiesto, che, per un microcontroller, non è mai troppo complicato da realizzare.

Questo è possibile dato che la disponibilità dell'hardware comprende una importante funzione: l'emulazione diretta on-chip attraverso il **motore di debug ICD** ([www](#)) **integrato** in numerosi microcontroller; con l'emulazione on-chip si ha il controllo diretto del componente e la **verifica reale** delle risposte alle varie azioni del programma.

Non tutti i chip dispongono di ICD, ma sono comunque debuggabili utilizzando un debug header opportuno. Nelle esercitazioni saranno impiegati solo chip con ICD integrato, per evitare il costo aggiuntivo dell'header esterno.

Per quanto riguarda i supporti fisici, si potranno impiegare le numerose schede di sviluppo disponibili sul mercato. Oppure, con un fai-da-te, usando breadboard, millefori e simili.

Volendo cambiare parti del circuito, aggiungere o togliere componenti, la soluzione millefori è la meno adeguata; un poco meglio le breadboard



Su breadboard è importante **non utilizzare componenti recuperati da dissaldature**, i cui terminali potrebbero forzare le molle dei contatti, oltre a lasciare residui di stagno. Inoltre, il cablaggio va fatto nel modo più ordinato possibile, meglio se con fili di diversi colori, in modo da poter tenere sotto controllo le connessioni e poter effettuare con sicurezza eventuali modifiche.

Nella foto, una breadboard distribuita da [TME](#) ([www](#)), con la predisposizione per i due morsetti di alimentazione

Sia su breadboard che su millefori l'uso di cavi disordinati o troppo lunghi va limitato perchè potrebbero dare origini a connessioni instabili. Se si salda su millefori occorrerà disporre di zoccoli per i chip in modo da non rischiare di surriscaldarli ed avere nel contempo la possibilità di sostituirli con altri.

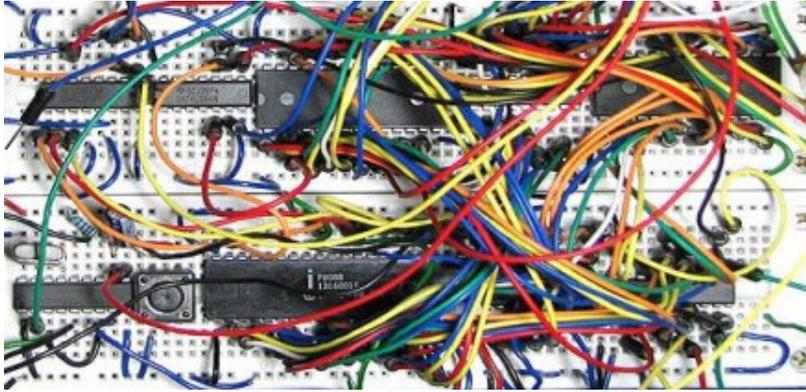


E' importante considerare che un cablaggio malfatto, una breadboard di scarsa qualità con contatti insicuri, fili volanti o saldature non ben eseguite sono causa di seri problemi che impediscono il corretto svolgimento del lavoro e possono essere non immediati da diagnosticare: si consuma inutilmente una gran quantità di tempo.

Se avete problemi, per prima cosa verificate con reale cura il vostro lavoro, perchè, **per esperienza,**

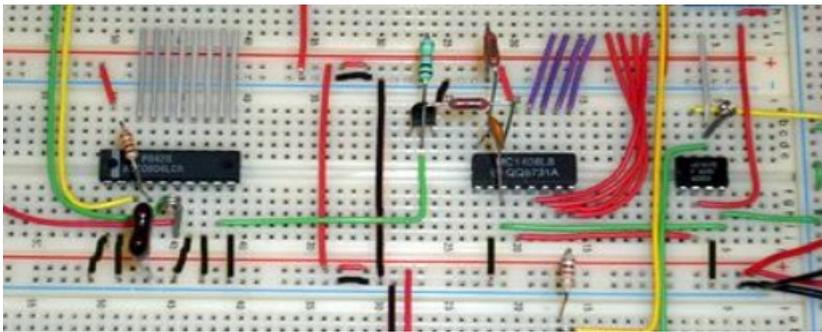
è praticamente sempre una scadente qualità dell'hardware che avete realizzato voi a impedire il funzionamento del software.

La scelta di breadboard o millefori, se è dettata da considerazioni economiche non è la scelta migliore, dato che ci si confronta con la necessità di impiegare anche molto tempo per realizzare un circuito decente.



Questo non è certamente il modo per realizzare un circuito su breadboard.

Il groviglio di cavi volanti, al minimo, impedisce qualsiasi modifica sui componenti al di sotto, mentre è certamente causa di mal funzionamenti.



Qui abbiamo una realizzazione decisamente migliore, con un minimo di componenti e cavi volanti.

Questo, però, richiede uno studio preliminare per ottenere la disposizione ottimale dei componenti e il minor groviglio nelle connessioni.

La non immediatezza nella realizzazione di un circuito funzionale sulla breadboard, a nostro parere rende molto più sensato **l'uso di schede di sviluppo, che forniscono una piattaforma stabile e sicura.**

Il mercato offre molteplici soluzioni, sia direttamente da Microchip che da altri costruttori.

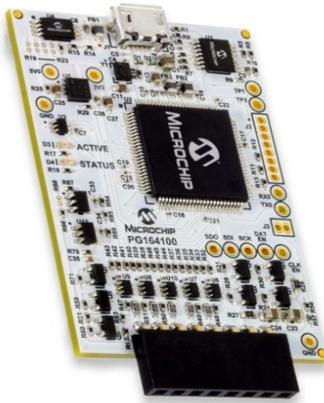


Le esercitazioni presentate qui si appoggiano alla scheda di sviluppo **LPCuB** (www), progettata per avere un supporto flessibile al massimo grado, sia per apprendimento che per prototipazione.

Supporta tutti i PIC in package 8/14/20 pin, integra la connessione ICSP e numerosi componenti esterni (LED, display, pulsanti, buzzer, potenziometro, sensore temperatura, ecc.) che sono collegabili con semplici jumper.

La necessità di avere tools adeguati alle nuove esigenze ha fatto sì che ICD2, PICKIT1, ICE2000 e ICE4000, che erano supportati da MPLAB, non lo siano più da MPLAB X.

Pur esistendo **cloni** di vario genere dei tools di Microchip, **non se ne consiglia l'acquisto**: troppe problematiche possono essere nascoste e la differenza di costo non è tale da compensare il rischio di trovarsi in mano un oggetto che funziona solo parzialmente.



Se il problema è la spesa, sono disponibili tools a bassissimo costo, come [SNAP](#) ([www](#)) (meno di 14€).

Anche se le prestazioni sono un poco minori di quelle dei PICKIT, si tratta di un prodotto molto recente (2019) che, nonostante l'economicità, consente il debug e la programmazione della maggior parte delle MCU flash PIC, dsPIC, AVR e SAM, utilizzando l'ambiente MPLAB X (ver. 5.05 o successive).

Come per gli altri tools, **MPLAB Snap** può essere collegato al PC tramite un'interfaccia USB 2.0 e al circuito in prova attraverso la connessione ICSP (In-Circuit Serial Programming).

Inoltre, c'è anche una scelta di schede di valutazione che **integrano una interfaccia di debug** ed il relativo collegamento all'USB del PC, come le board [Express](#) ([www](#)) e [Curiosity](#) ([www](#)) di Microchip, rendendo inutili tools esterni.



Una avvertenza: PICKIT4, SNAP, le board Express sono supportate dalle ultime versioni di MPLAB X, 5.05 e successive, e queste richiedono almeno Windows 7. Quindi, nella scelta, valutate tutti gli elementi, tenendo presente che, se cominciate da zero, è meglio utilizzare materiale recente piuttosto che datato o addirittura obsoleto.

Nel caso in cui vogliate fare sul serio ed il costo non è un limite invalicabile, ci sono tools con maggiori prestazioni



Se 200€ per un buon tool di sviluppo non è un limite invalicabile, ci sono gli ICD, nel caratteristico (ma discutibile) case rotondo.

In particolare l'[ICD4](#) ([www](#)) che è la versione più recente. La scelta può essere decisa dal continuo aggiornamento del firmware del dispositivo per seguire il costante progresso della produzione di nuovi chip e che hanno reso obsoleti ICD2 e ICD3.

Esistono, poi, altri hardware di sviluppo, come i prodotti di [Mikroelektronika](#) ([www](#)). Questi sono legati ad ambienti C diversi da XC8 e non saranno presi qui in considerazione, pur essendo oggetti molto ben realizzati e potenti.

Una nota per quanto riguarda il **“programmatore”**:

**E' purtroppo comune che chi si avvicina alla programmazione dei microcontroller indichi come principale priorità il possesso di un “programmatore”.
Invece, è proprio un oggetto che non serve per nulla!**

Non occorre alcun dispositivo di programmazione, in quanto, attraverso la connessione **ICSP** ([www](#)), possiamo usare gli stessi tool di sviluppo sia come debugger che come programmatori, all'interno dell'ambiente integrato MPLAB-X.

In particolare, la tendenza del neofita è quella di **auto costruirsi** uno dei tanti **xxxpippo** o JDM i cui schemi disponibili in rete.

Non è per nulla una buona idea, dato che mai il copiare pedissequamente un circuito dal web ha permesso di rendersi conto del modo in cui sono programmati i PIC (e i microcontroller in generale), mentre può esporre a cocenti delusioni. In particolare, sono da evitare le versioni dipendenti da un port RS232, sia perchè questo non è più presente su gran parte dei PC, sia perchè le tensioni fornite possono essere del tutto insufficienti per una corretta programmazione. Inoltre, mentre con i tools del costruttore dei chip si può avere supporto e assistenza e c'è una sicurezza della qualità dell'hardware e dell'aggiornamento del firmware, con i prodotti generici questo manca e i problemi che certamente vengono a sorgere possono rendere un hobby o un lavoro estremamente spiacevoli.

Quindi, se volete salvarvi da un gran numero di problemi e di tempo buttato, **evitate di sprecare denaro in quegli accrocchi che vengono proposti come “programmatore”**, in primo luogo quelli a “costo circa 0” o altri oggetti molto nominati, la cui utilità è praticamente nulla, anche perchè hanno pochissime probabilità di essere aggiornati per i nuovi chip che vengono prodotti.

Con una connessione ICSP e uno dei tools originali NON serve alcun altro tipo di “programmatore”. E, fate bene i conti, il materiale originale non costa molti di più (e a volte, di meno), dei cloni.

Se non volete astenervi da una auto costruzione, puntate almeno sui cloni di PICKT2 e PICKIT3 che sono disponibili sul WEB: la realizzazione di questi circuiti, non semplici, qualcosa può darsi che insegni e permette di avere tra le mani un oggetto con qualche utilità (anche se il costo finale della realizzazione è probabile che non sia tanto lontano da quello dell'originale).

Non stiamo consigliando di abbandonare le auto costruzioni, tutt'altro! Consigliamo, però, di lasciare la costruzione di **strumenti di sviluppo** a quando si avrà una sufficiente conoscenza dei chip e delle loro caratteristiche (ed allora apparirà chiara la motivazione di questa opposizione a cloni e accrocchi vari).

Servirà, invece, un **sistema di alimentazione**. Gli Enhanced Midrange che usiamo in queste esercitazioni sono disponibili in due versioni: F con alimentazione da 2.3 a 5,5V e LF con alimentazione da 1.8 a 3.6V.

Si potranno usare alimentatori da laboratorio, alimentatori fissi, accumulatori, batterie (**ma non caricabatterie!!!**); ad esempio tre pile AA, da cui si ottengono 4.5V, vanno bene. Anche i 5V

ottenibili da una presa USB sono adatti.

La corrente necessaria per le esercitazioni non è grande e una capacità di qualche centinaio di milliamperè è sufficiente. Da notare che, attraverso la connessione ICSP, non solo possiamo **programmare il chip on-board**, ma possiamo anche [alimentare la scheda di prova](#) ([www](#)) quando usiamo i PICKIT.

Nella scelta dei chip, sono stati privilegiati i componenti della **famiglia Enhanced Midrange** ([www](#)) perchè sono quelli attualmente soggetti a maggiore sviluppo, assieme ai PIC18F.

I Baseline e i Midrange sono famiglie ormai non più aggiornate, ma si possono comunque impiegare, anche se sono da considerare obsolete e non adatte a nuovi progetti.

Occorre fare alcune considerazioni su questo argomento che sembra particolarmente difficile da digerire a molti, principalmente in ambiente hobbistico.

La principale riguarda i PIC che hanno un “nome” dovuto alla loro elevata diffusione. Sigle come 16F84, 16F877, 12F625, 16F628 sono ben note perchè questi chip sono stati, ai loro tempi, oggetto di ampia sperimentazione e il web è pieno di articoli e corsi basati su di essi. Ma se andiamo a vedere quando queste pagine sono state scritte, ci troviamo di molti anni nel passato. Nel frattempo la tecnologia non si è certo fermata.

Quindi, va detto con chiarezza che è **poco utile insistere su componenti datati**, che hanno prestazioni e risorse non comparabili con quelli attuali, oltre ad avere un prezzo assai maggiore. Anche se li avete nel cassetto, riservateli per qualche applicazione specifica.

Questo non toglie che non sia possibile applicare le esercitazioni, con le dovute modifiche, anche a questi cari vecchietti, ma, insistendo ad utilizzarli, va perso qualsiasi contatto con lo stato attuale della tecnologia.

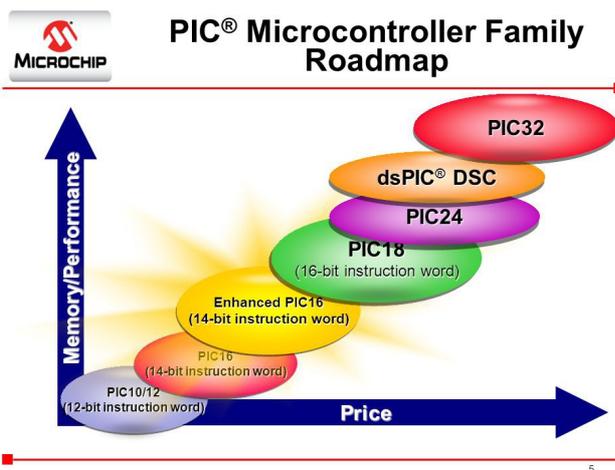
Questa tabella comparativa permette di avere una idea delle differenze tra le varie famiglie di PIC a 8 bit.

	Baseline	Midrange	Enhanced Mid.	PIC18F
Pin	6 - 40	8 - 64	8 - 64	18 - 100
Interrupt	no	singolo	Singolo con auto context save	Multiplo con auto context save
Performances	5 MIPS	5 MIPS	8 MIPS	16 MIPS
Istruzioni	33 a 12 bit	35 a 14 bit	49 a 14 bit	83 a 16 bit
Program memory	max. 3K	max. 14K	max. 28K	max. 128K
Data memory	max. 138 bytes	max. 368 bytes	max. 1.5 kB	max. 4 kB
Stack	2 livelli	8 livelli	16 livelli	32 livelli
Uso con il C	non ottimizzati	non ottimizzati	ottimizzati	massima ottimizzazione

Dato che il corso è focalizzato sull'uso del C, è evidente che la scelta cade su componenti che sono previsti per supportare al meglio questo linguaggio e le sue esigenze (molta RAM, ampio stack, set di istruzioni adeguate, ecc.).

Per il Corso Assembly sono stati utilizzati ampiamente Baseline e Midrange, ma, anche se è possibile scrivere programmi in C per questi processori, non è la scelta migliore, date le loro limitatissime risorse ed il fatto che sono considerati obsoleti e da tempo non sono previsti ampliamenti dei modelli.

Ci si può chiedere perchè non si è partiti con i PIC18F, che sono il top della produzione Microchip a 8bit e che da sempre è la famiglia a 8 bit dotata delle migliori soluzioni per supportare una programmazione in C. Attualmente, però, con l'avvento degli Enhanced Midrange, si sono allargate le possibilità pratiche di usare il C anche con PIC16F.



Da un certo tempo Microchip ha focalizzato gli sforzi di sviluppo di nuovi prodotti su questa famiglia, che ha il vantaggio di un costo costruttivo minore dei PIC18F e che, in pratica, costituiscono un ponte tra i vecchi modelli e i più performanti nel rapporto tra prezzo e prestazioni.

Peraltro, l'uso del C permetterebbe un approccio diretto anche ai PIC24 o ai PIC32, ma sarebbe più complesso quello verso i loro hardware, che, per chi affronta la programmazione di embedded, è un elemento fondamentale.

[Una introduzione agli Enhanced Midrange qui \(www\)](#)

Si è pensato di utilizzare chip che sono una conseguenza dei Midrange, probabilmente già noti a chi legge queste pagine, e quindi un passaggio dal vecchio al nuovo quasi indolore, mentre si scoprono miglioramenti e prestazioni interessanti, a costi quanto mai bassi.

Con un prodotto allo stato dell'arte avete la possibilità di imparare ad usare nuove e potenti periferiche: Enhanced Midrange sono pin-to-pin compatibili con i PIC precedenti, offrono un set di istruzioni esteso e una notevole varietà di periferiche.

Hanno disponibilità di memoria e frequenza di clock molto maggiori dei Midrange e ne risolvono un gran numero di problematiche, **rendendo la vita molto più facile** al programmatore e fornendo prestazioni fino al 50% maggiori, con una riduzione delle linee di codice anche del 40%. Fatto importante, l'ampia disponibilità di memoria, la diversa struttura dello stack, il set di istruzioni ampliato e le funzioni aggiunte fanno sì che gli Enhanced si prestino bene a supportare programmazione in C molto meglio dei Midrange e dei Baseline. Dispongono di motore ICD integrato che facilita l'emulazione on circuit e di nuove periferiche core indipendenti che, per ora, non sono apparse nei PIC18F. I componenti della famiglia Enhanced Midrange sono molto più uniformi tra loro (era ora...) che non i vecchi Midrange, il che rende meno difficoltoso il passaggio da un chip ad un altro. Inoltre, non ultimo, tutto questo ha un costo minore, e anche molto minore, dei vecchi modelli obsoleti.

Qui trovate [pagine di descrizione della famiglia Enhanced](#) ([www](#))

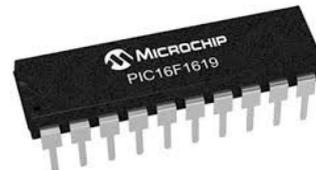
Certamente l'uso dei PIC18F permetterebbe un ancora migliore supporto al C, ma, come detto, alcune nuove periferiche, alla data di stesura di queste pagine, non sono ancora apparse in questi PIC e funzioni come acceleratore matematico, zero cross detector, angular timer, ecc. possono rivelarsi risolutive per varie applicazioni.

Altri chip, anche gli antichi 16F877 o 12F625 saranno facilmente impiegabili, adeguando il sorgente, dove necessario, alle loro caratteristiche hardware, ma se usiamo chip datati è possibile che non ci sia la possibilità di eseguire tutte le esercitazioni, per mancanza di risorse.

Quindi, nulla toglie che si possano scrivere programmi in C anche per i primitivi Baseline o continuare a usare PIC16F877 per situazioni molto particolari.

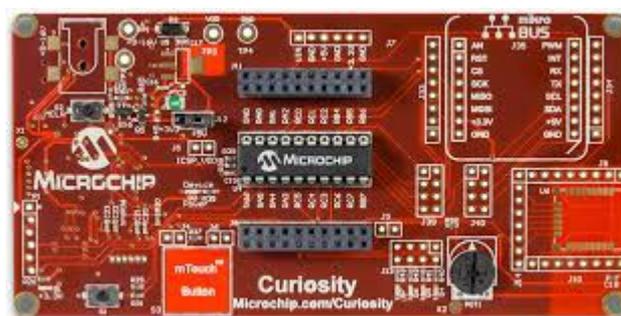
Ma, certamente, non ha proprio senso per chi comincia: pensate sia più utile e più semplice imparare a guidare su una Ford modello T piuttosto che su un'automobile attuale?

Le nostre esercitazioni si svolgeranno utilizzando chip a 8 pin, ad esempio [12F1571/2](#), e chip a 20pin, come [16F1619](#).



Questi chip esistono anche in altri packages per surface mount, ma è evidente che le versioni DIP sono quelle più adatte per esercitazioni, prototipi, usi generali, sia per hobbisti che per professionisti.

Tra l'altro, PIC16F1619 è abbastanza comune: lo trovate anche installato su una delle [Curiosity board](#) ([www](#)), scheda di valutazione e sviluppo di Microchip.



Questa scheda, come altri prodotti di Microchip, può essere un utile supporto per le esercitazioni, con un costo limitato.

Comunque, non è certo il costo del chip a poter determinare la scelta di un hobbista: buona parte dei piccoli microcontroller costa meno di 2€ e anche meno di 1€.

Per un confronto tra vecchi e nuovi PIC che dovrebbe togliere ogni dubbio:

	16F84A	12F675	12F1572	16F1619
RAM	68	64	256	1k
Flash	1k	1k	2k	8k
Clock	20MHz solo esterno	20MHz interno/esterno	32MHz interno/esterno	32MHz interno/esterno
Istruzioni	35		49	
Pin – I/O	18 - 13	8 - 6	8 - 6	20 - 18
Timer	1	2	4	7
Comparatori	-	1	1	2
PWM 16bit	-	-	3	4
ADC 10 bit	-	4 canali	4 canali	12 canali
DAC	-	-	1 - 5bit	1 – 8bit
Altri	-	-	EUSART, CWG, Temp. , FVR, XLP, ecc.	+ CRC, ZCD, CLC, SMT, HLT, PID, MSSP, 100mA I/O
ICD	no	no	si	si
Costo	5.75€	0.90€	0.67€	1.06€

Costi su MicrochipDirect alla data di stesura della pagina.

Dovendo acquistare ex novo i microcontroller, è consigliabile scegliere **il top di una famiglia**. La differenza di costo è minima. Se abbiamo in mano 12F1572 possiamo sfruttarne tutte le possibilità; quando, poi, si realizzasse una applicazione commerciale dove il costo è un parametro importante, e che non si richiedono questi plus, il passaggio a 12F1571, meno dotato, ma meno costoso, è immediato. Così, nella serie 16F1615/9 sarà preferibile sviluppare su 16F1619 che dispone del massimo delle opzioni per poi passare ad uno dei chip inferiori della famiglia, dove sono necessarie minori risorse.

Anche l'apprendimento non è per niente ostacolato dal volume delle nuove caratteristiche. In particolare, se un 12F675 dispone di un manuale di 130 pagine, questo non vuol dire che sia “più semplice” da usare di un 12F1572 il cui foglio dati occupa 334 pagine. In effetti, **l'utilizzatore, a parte le consistenti differenze, trova molto più semplice, e con un minore numero di sorprese, usare il secondo piuttosto che il primo.**



La quantità di pagine del foglio dati è relativo essenzialmente alla quantità di funzioni e periferiche disponibili, ma se non usiamo queste funzioni e periferiche, nella pratica è come se non esistessero

Semplicemente, dato che all'avvio saranno disabilitate per default, possiamo non considerarle. Leggeremo i relativi capitoli del manuale non appena avremo necessità di usarle.

In ogni caso, il C costituisce una discreta astrazione con l'hardware e, tenendo conto delle caratteristiche di base di ogni chip, il passaggio da un componente all'altro è più semplice che usando l'Assembly.

Vediamo ora di aggiungere un “minimo” di informazioni che dovrebbero essere sufficienti per iniziare sull'ambiente Microchip e sul compilatore XC8.

E' possibile che conosciate già quanto verrà detto nelle pagine seguenti e, in tal caso, queste pagine potranno essere saltate, ma è anche possibile che non siate al corrente di tutto e, in tal caso, è opportuno leggerle.

Forniscono gli elementi minimi per potersi orientare nelle esercitazioni.

0.2-Struttura dei file del corso

Le esercitazioni sono realizzate con MPLAB X 4.13 o successivi, XC8 1.34 o successivi.



Si raccomanda di mantenere aggiornata la versione di MPLAB X e del compilatore XC8 alle ultime disponibili sul sito di Microchip in quanto è possibile che una versione più recente corregga problemi che si presentavano in quella precedente, oltre ad offrire maggiori prestazioni.

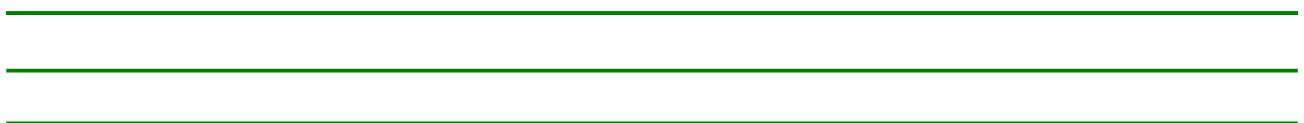
Non è necessario cancellare versioni precedenti: non c'è alcun conflitto o problema, se non di occupazione di spazio sul disco rigido, nel fatto di avere più versioni di questo software.

Le esercitazioni sono accompagnate dai relativi progetti, previsti per essere conservati sul disco rigido nella directory *C:\Corso_C*.

Ogni esercitazione dispone di una cartella *esxC* dove *x* è il numero dell'esercitazione.

Una cartella *C:\Corso_C\MyIncludes* contiene file accessori necessari alle esercitazioni.

E' possibile utilizzare qualsiasi altra struttura si ritenga migliore o più adeguata alle proprie esigenze, modificando i path dove necessario.





Le pagine seguenti contengono informazioni generali sugli elementi che si incontreranno nelle esercitazioni.

E' possibile che siate già al corrente di quanto trattato e quindi queste pagine sono superflue.

Se, invece, avete poca dimestichezza con i PIC può essere utile leggerle.

Inoltre sono riportate alcune informazioni basilari sul compilatore C usato.



0.3 - Alcuni termini comunemente usati trattando con i PIC.

Elenchiamo alcuni termini comunemente usati nell'ambiente Microchip.

-
- **ADC:** Analog to Digital Converter ha lo scopo di trasporre in digitale un segnale analogico. Alcuni PIC dispongono di modulo ADC integrato con risoluzione tra 8 e 12bit e con vari canali di ingresso
- **Alfanumerico:** insieme di caratteri che comprendo quelli alfabetici e quelli numerici
- **ALU:** Arithmetical Logic Unit – unità logico aritmetica – è la parte di un elaboratore che tratta le operazioni logiche (and, or, ecc) e quelle aritmetiche (somma, sottrazione, ecc)
- **Analógico:** che supporta dati di tensione , tipicamente di valore variabile tra la Vss e la Vdd
- **ANSI:** American National Standards Institute è un'organizzazione responsabile della formulazione e dell'approvazione degli standard negli Stati Uniti.
- **Assembler:** Uno strumento che traduce il codice sorgente dell'Assembly (.asm) in codice macchina. MPASM è l'assemblatore di Microchip
- **Assembly:** Un linguaggio simbolico che descrive il codice binario della macchina in una forma leggibile.
- **Banchi:** a causa della limitata lunghezza delle istruzioni (opcodes) del microcontoller, non c'è spazio per contenere l'intero indirizzo di tutte le locazioni della RAM. I PIC adottano, quindi, un sistema che richiede uno o due bit esterni per completare l'indirizzamento. Ne risulta che, dal punto di vista logico, la RAM è divisa in più banchi (negli Enhanced Midrange i banchi sono 32). Quindi, per accedere ad un registro, occorre indirizzare sia la locazione voluta che il banco in cui si trova. Fortunatamente il compilatore C dispone di un automatismo che libera l'utente dalla necessità di commutare i banchi, cosa che invece è necessaria in Assembly.
- **BOR (Brown Out Detect):** un modulo che blocca il funzionamento del microcontroller e lo resetta se la tensione di alimentazione scende al di sotto di un limite programmato, per evitare errori di funzionamento
- **C11, C99:** sono diverse revisioni dello standard emesse dall'ISO (International Organization of Standards); l'ente ha rilasciato quattro versioni del linguaggio C, note come C95 (ISO/IEC 9899/AMD1:1995), C99 (ISO/IEC 9899:1999), C11 (ISO/IEC 9899:2011/Cor 1:2012) e C18 (ISO/IEC 9899:2018).
- **Clock:** il segnale sul quale si sincronizzano le operazioni del microrontroller.
- **Context saving** (salvataggio del contesto) : è una operazione necessaria prima dell'ingresso nella routine di gestione dell'interruzione, a cui fa seguito, all'uscita, il restore di quanto

salvato. Il salvataggio del contesto comprende lo spostamento manuale (da programma...) in locazioni RAM dei core registers (STATUS, PCLATH, W per i Midrange), il che richiede varie istruzioni e occupa alcune locazioni della RAM dati. L'operazione inversa va fatta all'uscita dall'interrupt per ricollocare al loro posto i registri.

Per gli Enhanced (come per i PIC18F) l'operazione di context saving & restore è del tutto automatica e non richiede alcuna istruzione, oltre a salvare i registri (PCL, STATUS, FSR0/1, BSR, WREG, PCLATH) in una area di RAM dedicata.

- **Core Independent Peripherals:** sono progettate per gestire le loro attività senza supervisione dalla CPU per mantenere il funzionamento. Di conseguenza, semplificano l'implementazione di sistemi di controllo complessi e offrono ai progettisti una ampia flessibilità di impiego.
- **DAC:** Digital to Analog Converter, svolge il lavoro opposto al precedente. Alcuni pic dispongono di modulo DAC integrato con risoluzione tra 5 e 10 bit.
- **Default al POR :** i valori forzati nei registri SFR dall'azione del modulo di reset di power on
- **Digitale:** che supporta dati binari 1-0
- **EUSART** (Enhanced Universal Asynchronous Receiver Transceiver) è una periferica per la gestione delle comunicazioni seriali e può essere configurata come un sistema asincrono o sincrono.
- **Flash:** memoria che conserva il suo contenuto anche in mancanza di alimentazione. Solitamente usata per conservare il programma (memoria programma).
- **Fosc:** il clock principale, definito in C come `_XTAL_FREQ`
- **Fosc/4 :** il clock delle istruzioni che richiedono per l'esecuzione 4 impulsi del clock principale.
- **Hardware:** (formalmente roba dura, ferramenta) è l'insieme dei componenti fisici che consentono il funzionamento di un computer o un insieme elettronico.
- **Harvard :** è la filosofia costruttiva dei PIC, per la quale il bus dati e il bus indirizzi sono separati e possono avere ampiezze diverse.
- **ICD** (In Circuit Debug) è un motore interno al chip che permette, attraverso la connessione ICSP, il debug on chip con funzioni come break point, step-by-step, ecc, senza altri dispositivi al di fuori di uno dei tools di sviluppo
- **ICSP** (In Circuit Serial Programming): connessione che interessa tre pin del microcontroller (**Vpp**, **ICSPDAT**, **ICSPCLK**) e che permette la programmazione on-board
-
- **Interrupt:** l'azione di una periferica che re indirizza il Program Counter al Vettore di Interrupt. [Altre informazioni su interrupt](#) ([www](#))

- **LED** (Light Emitting Diode) : un componente (diodo) che produce luce se attraversato da corrente. Ha una tensione di funzionamento tipica, tra 1.8 e oltre 4V, a seconda del colore, ed una corrente tipica tra 1 e 20mA, a seconda del modello. LED per illuminazione hanno consumi di corrente molto maggiori e, per essere gestiti dal microcontroller, richiedono un driver di potenza adeguato.
- **Literals** – sono valori numerici. *Integer literals* sono numeri interi con o senza segno
- **Moduli**: sono chiamati così i blocchi hardware all'interno del microcontroller che eseguono una funzione specifica. Ad esempio, il generatore di tensione di riferimento FVR, il convertitore AD, i timer, ecc, sono chiamati moduli.
- **Oscillatore**: un dispositivo che genera un segnale (**clock**) che sincronizza le varie operazioni dei circuiti interni al microcontroller.
- **Pagine**: la memoria programma (Flash), a causa della limitata lunghezza delle istruzioni (opcodes) del microcontroller, è suddivisa in pagine se supera 2k. La gestione delle pagine (paging) è effettuata con diversi metodi che modificano il Program Counter. Il compilatore C dispone di un automatismo che libera l'utente dalla necessità di commutare le pagine, cosa che invece è necessaria in Assembly.
- **Periferiche**: (*peripherals*) un altro nome per chiamare i blocchi che eseguono una determinata funzione che, in generale, ha un rapporto con l'esterno. Sono periferiche i port, le interfacce analogiche, i timer, le comunicazioni seriali, ecc.
- **POR (Power On Reset)** : è il reset che avviene all'arrivo della tensione di alimentazione. Al POR, nei registri SFR vengono caricati dei valori di default specifici.
- **PORT**: un registro SFR che corrisponde ad un blocco di pin usati come I/O digitali, organizzati in gruppi da 1 a 8 bit e identificati con lettere (PORTA, PORTB, PORTC, ecc), in quantità variabile in dipendenza dai pin del chip.
- **PWM** : Pulse Width Modulation - genera un segnale a frequenza fissa e larghezza di impulso variabile, fondamentale in numerose applicazioni
- **RAM**: formalmente Random Access Memory, è una area di memoria il cui contenuto si cancella se viene a mancare l'alimentazione. Solitamente utilizzata per conservare dati durante il funzionamento del programma. Per i PIC sono in area Ram i registri dati e gli SFR.
- **Registro**: un insieme tipicamente di 8bit che occupa un indirizzo nella mappa di memoria RAM. Comprende sia RAM dati, sia registri di controllo delle periferiche.
- **Reset** : è l'azione di un modulo interno che riporta gli SFR ad una condizione di default nota. Negli Enhanced esistono numerose cause di reset:
 - Power-on Reset (POR)
 - Brown-out Reset (BOR)
 - Low-Power Brown-out Reset (LPBOR)

- MCLR Reset
- WDT Reset
- istruzione RESET
- Stack Overflow e Stack Underflow
- Programming mode exit

Un capitolo apposito del foglio dati (nel caso di 12F1571/2 è il cap. 6) raccoglie tutto quello che c'è da sapere sui reset e le condizioni di default relative

- **RISC:** (*Reduced Instruction Set Computer*) i PIC sono RISC, ovvero dispongono di un ristretto set di istruzioni in cambio di una maggiore efficienza nell'esecuzione
- **SFR (Special Function Register):** sono i registri che controllano i moduli integrati (ved. più avanti). In XC sono identificati con label pre assegnate, scritte in lettere maiuscole. Ad esempio: **PORTA**, **STATUS**, **OSCCON**, ecc.
- **Timer:** una periferica dotata di un contatore e pensata specificamente per supportare temporizzazioni o conteggi (timer counter).
- **Vettore di Interrupt:** la locazione di memoria programma alla quale il Program Counter punta dopo una chiamata di interrupt. Nei PIC a 8 bit si trova all'indirizzo 0004h per quelli dotati di un solo livello di interruzione; a 0004h e 0008h per quelli dotati di interrupt a due livelli (PIC18F).
- **Vdd:** la tensione di alimentazione positiva. In generale per i PIC va da 2.3 a 5.5V per i modelli F. Per quelli LF va da 1.8 a 3.6V.
- **Vettore di Reset:** la locazione di memoria programma alla quale il Program Counter punta dopo un Reset. Nei PIC a 8 bit si trova all'indirizzo 0000h.
[Altre informazioni su Reset](#) ([www](#))
- **Vss:** la tensione di alimentazione negativa o massa (GND)

Voci relative ai compilatori:

- **Compilatore:** una applicazione che trasforma il file sorgente nel file eseguibile dal processore
- **Errors:** gli errori segnalano problemi che **rendono impossibile continuare la compilazione** del programma. Quando possibile, gli errori identificano il nome del file di origine e il numero di riga in cui il problema è evidente.
Da sapere che spesso la segnalazione di un errore è riferita alla riga successiva a quella dove l'errore è presente. Inoltre, una catena di errori solitamente dipende da un singolo errore iniziale, corretto il quale la segnalazione cessa.
- **File eseguibile o file hex:** il file binario generato dalla compilazione e che contiene i codici che dovranno essere eseguiti dal microcontroller
- **File sorgente o codice sorgente:** è il testo che contiene le istruzioni che dovranno essere

compile. E' scritto in un linguaggio di programmazione (Assembly, C, BASIC, Pascal, ecc), rispettando le regole del linguaggio.

- **ICD**: in circuit debug è una funzione che permette di debuggare l'applicazione direttamente dal chip che la esegue.
- **IDE: integrated development environment** è una applicazione che supporta il programmatore durante le fasi dello sviluppo del codice sorgente (debugging), integrando i compilatori, le utility di programmazione, le simulazioni e quanto altro necessario.
- **Software**: (neologismo americano, formalmente “roba morbida”) è l'insieme delle componenti immateriali di un sistema programmabile
- **Warnings**: gli avvisi (warnings) riportano condizioni che potrebbero indicare un problema, **ma non interrompono la compilazione**. Una compilazione corretta non dovrebbe presentare warnings. Sta all'utente valutare se la segnalazione è un eccesso di zelo del compilatore oppure si tratta di qualcosa che può mettere a rischio il risultato della compilazione

Inoltre:

- E' uso comune indicare un gruppo di bit con la forma **<X:Y>**.
Ad esempio **bit<4:0>** comprende tutti i bit da 0 a 4.
- Nell'indicare livelli logici, si possono usare varie espressioni equivalenti:

0	basso	low	FALSO	clear	≈Vss
1	alto	high	VERO	set	≈Vdd

- Per Microchip, **set** è portare a 1 e **clear** è portare a 0
- Il compilatore XC impiega la disposizione dei dati nel formato [little endian](#) ([www](#))
- Tutte le label pre definite che indicano registri SFR sono, per XC8, solo in lettere maiuscole. Dato che XC è case sensitive, **lata** non sarà la stessa cosa di **LATA**.

0.4 - Gli I/O negli Enhanced Midrange.

Le esercitazioni iniziali si svolgono essenzialmente attorno al comando degli I/O digitali. Ora, non è possibile agire in modo sensato sull'hardware se non se ne conoscono almeno le caratteristiche essenziali.

Per questa ragione aggiungiamo almeno una pagina sull'argomento, rimandando per altre informazioni più dettagliate alle pagine specifiche della famiglia **Enhanced Midrange** ([www](#)).

Strutturalmente, gli **Enhanced Midrange**, sono, come dice il nome, una evoluzione dei ben noti Midrange, di cui mantengono molte caratteristiche, ma a cui ne aggiungono molte di nuove.

Sono da considerare alcune caratteristiche generali dei PIC:

1. nei microcontroller, ogni pin può assumere numerose funzioni, tra cui quella di I/O digitali.
2. All'accensione, per ragioni di basso consumo, è probabile che ai pin siano attribuite funzioni di ingresso analogico. Per poter accedere alle altre funzioni occorre disabilitare questi ingressi analogici da programma.
3. Praticamente tutti i pin degli Enhanced Midrange possono essere usati come ingressi o uscite digitali; con questo si intende che in uscita il pin può assumere i livelli logici 1 o 0, seguendo i comandi del programma, con la possibilità di gestire una certa corrente (mediamente per i PIC si parla di 25mA).
4. I bit di controllo dei pin sono raccolti in blocchi chiamati PORT, a 8 bit; ogni bit corrisponde ad un pin, numerati da 0 a 7.

Per quanto riguarda gli I/O, per gli **Enhanced Midrange sono disponibili tre registri per ogni pin, similmente ai PIC18F:**

- registro **PORTx**, che si comporta come quello degli altri PIC
- registro **TRISx**, per la direzione, analogo agli altri PIC
- registro **LATx**, che permette di scrivere e leggere il contenuto del latch del port, similmente ai PIC18F

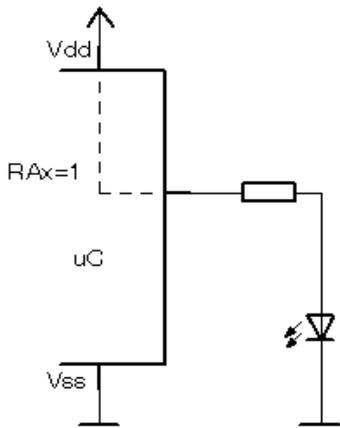
Questi registri sono leggibili e scrivibili da programma e sono identificati con label tipiche predefinite.

Ricordiamo che Baseline e Midrange dispongono solo dei registri **PORT** e **TRIS**.

Per maggiori dettagli, il funzionamento di un port è descritto [qui](#) ([www](#)).

Ci sono altre importanti caratteristiche degli I/O, ma in questa prima esercitazione non sono coinvolte e le vedremo con il procedere del corso.

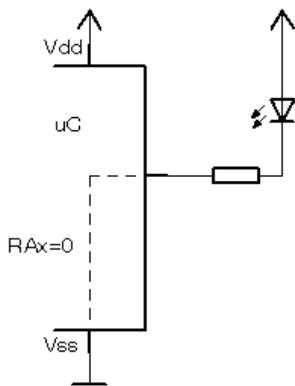
Per dare qualche altro dettaglio, possiamo dire che, quando programmiamo il livello logico di un pin configurato come uscita digitale, in pratica facciamo sì che il buffer interno a monte del pin lo connetta con la Vdd, ovvero con la tensione di alimentazione positiva. Attraverso questo collegamento potranno essere erogati (source current) mediamente fino a 25mA (50mA per gli Enhanced), quindi una corrente più che adeguata per alimentare un LED (corrente tipica da 1 a 20mA a seconda del modello).



Volendo comandare un LED collegato tra pin e Vss (gnd), abbiamo una logica "diretta": il LED è acceso quando il livello del pin è 1, ovvero al buffer di uscita è connessa la tensione positiva Vdd.

Livello logico	Tensione al pin	LED
0	Vss	spento
1	Vdd	acceso

Se programmiamo l'uscita per un livello logico 0, il buffer interno conetterà il pin alla Vss; per i PIC, anche in questo caso, la corrente assorbibile (sink current) sarà di 25mA tipici (50mA per gli Enhanced).



Volendo comandare un LED collegato tra pin e Vdd (+ dell'alimentazione), abbiamo una logica "inversa": il LED è acceso quando il livello del pin è 0, ovvero al buffer di uscita è connessa la massa Vss.

Livello logico	Tensione al pin	LED
0	Vss	acceso
1	Vdd	spento

La scelta di un collegamento piuttosto che dell'altro dipenderà dalle necessità circuitali ed è considerata del tutto indifferente per il microcontroller: è compito del programma agire correttamente a seconda dell'hardware.

Se colleghiamo il LED alla Vss, l'accensione del LED è attivata dalle seguenti azioni del programma:

1. impostare il pin che comanda il LED come uscita digitale
2. portare il livello del pin a 1

Va notato con attenzione che **i pin dei PIC possono assumere molte funzioni differenti** ed all'accensione, possono non essere immediatamente disponibili come I/O digitali.

Quindi il punto 1 può richiedere interventi su diversi registri.

Tipicamente, **per disporre di un pin come uscita digitale può essere necessario:**

- non impostare funzioni opzionali che occupino quel pin dalla Configuration Word (ad esempio, CLKOUT)
- escludere le funzioni analogiche (registri ANSELx)
- portare a 0 il bit relativo del registro TRISx

Negli Enhanced Midrange è possibile anche determinare:

- se il pin in uscita è open drain o push-pull (registro ODCONx)
- se lo slew rate è normale o limitato (registro SLRCONx)

Da notare che chip Enhanced Midrange possono disporre di alcuni pin in grado di trattare correnti sink/source fino a 100mA. Questa opzione viene abilitata agendo da programma su un registro apposito.

Per disporre di un pin come ingresso digitale può essere necessario:

- non impostare funzioni opzionali che occupino quel pin dalla Configuration Word (ad esempio, CLKOUT)
- escludere le funzioni analogiche (registri ANSELx)

La direzione ingresso è selezionata per default nei registri TRIS all'accensione.

Negli Enhanced Midrange è possibile anche determinare:

- se il pin in ingresso è TTL o ST (registro INLVLx)

0.5-Le memorie dei PIC.

Sempre nell'ottica di disporre di una conoscenza anche minima delle caratteristiche dei componenti che andiamo a programmare, vediamo qualche parola sulla memoria, anche se sarebbe meglio parlare di memorie, dato che all'interno del chip ne esistono di diverse tipologie.

Nei PIC sono disponibili due aree di memoria distinte: una area di memoria programma e un'area di memoria dati, dato si tratta di strutture [Harward](#) (www).

Dati e istruzioni si trovano scritti in due aree diverse della memoria del componente, diverse anche come tecnologia: i dati sono in RAM CMOS, volatile, mentre le istruzioni sono in memoria Flash, non volatile.

La **memoria programma** è realizzata in tecnologia Flash; conserva i dati scritti per un tempo valutabile in molte decine di anni. Essenzialmente contiene l'eseguibile sotto forma di byte binari. L'ampiezza di questi "bytes" è in realtà variabile da 12 a 16 a seconda della famiglia PIC presa in considerazione. La flash può essere scritta e modificata durante la programmazione; i chip attuali hanno la possibilità di accedere alla Flash in lettura e scrittura anche da istruzioni durante l'esecuzione del programma.

La **memoria dati** è realizzata in tecnologia RAM CMOS; può essere letta e scritta da programma, ma conserva il suo contenuto solo fino a che è presente la tensione di alimentazione. Per i PIC a 8 bit, questa è l'ampiezza del dato (ci sono altri microcontroller in grado di trattare dati a 16 e 32 bit).

Nei PIC troviamo anche una terza area, denominata in passato **EEPROM** (Electrically Erasable Programmable Read Only Memory) utilizzata dal programma per conservare dati non volatili. Attualmente questa area è implementata in una zona della Flash ad alta affidabilità.

I bus di accesso alle due aree di memoria sono separati e possono essere di dimensione diversa. Nei PIC che trattiamo qui, il bus dati è ampio 8 bit, ovvero una locazione di RAM è a 8 bit (8 bit formano un byte). Per questo si parla di processori a 8 bit. Altre famiglie hanno dati a 16 e 32 bit.

Il bus delle istruzioni, ovvero quello della memoria programma, va da 12 bit per i Baseline a 14 bit per i Midrange e a 16bit per i PIC18F; un "byte" in questa area sarà quindi ampio da 12 a 16 bit a seconda del componente utilizzato.



Dunque, quando parliamo di PIC a 8 bit intendiamo quei componente che trattano dati a 8 bit, ma che possono avere istruzioni codificate su 12, 14 o 16 bit.

Ricordiamo che i PIC sono basati sull'[architettura Harvard](#) (www) per la quale bus dati e bus istruzioni sono separati e possono avere dimensioni differenti.

Una zona dell'area flash che conserva il contenuto fino ad una successiva riprogrammazione è quella delle opzioni di **configurazione iniziale**.



E' importante comprendere che **la configurazione iniziale non ha nulla a che fare con le istruzioni del programma.**
Le righe della configurazione iniziale non originano alcuna istruzione

Quando inseriamo nel sorgente le linee dedicate alla configurazione, queste non originano alcuna istruzione; indicano solo al compilatore di informare il dispositivo di programmazione che le locazioni dedicate ai registri di configurazione dovranno essere programmate con determinati valori.

La configurazione iniziale ha lo scopo di modificare le opzioni che l'hardware del microcontroller offre, per adattarsi alle esigenze di funzionamento del programma.

In questa area, detta **Configuration Word**, che negli Enhanced occupa comunemente 2 o più "bytes", si stabilisce come sarà l'hardware interno al chip, decidendo, prima dell'avvio del programma, quale genere hardware sarà disponibile.

Queste predisposizioni dell'hardware del microcontroller devono essere attive per poter supportare l'applicazione. Tra le più importanti ricordiamo:

- la scelta dell'oscillatore del clock primario
- l'attivazione di **MCLR** o meno
- l'attivazione di **WDT** (WatchDog Timer), **PWRT** (PoWeR on Timer), ecc.

Ad esempio, è possibile scegliere se il clock dipenderà da un oscillatore interno o esterno, se un pin sarà dedicato a un reset esterno o sarà un ingresso digitale, ecc.

Sono, per loro natura, in gran parte non modificabili da programma, anche se la tendenza attuale di progetto per gli Enhanced è il permettere la modifica di diverse di queste opzioni anche durante l'esecuzione delle istruzioni, in modo da concedere al programmatore più possibilità per far fronte alle possibili esigenze dell'applicazione.

0.6 - Gli SFR negli Enhanced Midrange.

Conoscere le possibilità del microcontroller che si sta usando è essenziale per poterlo utilizzare al meglio. In particolare, le periferiche integrate e le funzioni essenziali del processore, che sono governabili attraverso alcuni registri su cui il programma andrà ad agire.

Microchip definisce **SFR** (*Special Function Register*) delle locazioni nell'area RAM (quindi leggibili e scrivibili da programma) il cui scopo è controllare i moduli interni del microcontroller. Ci sono SFR per gli I/O, per l'EUSART, i comparatori, gli oscillatori, i timer, l'ADC, il PWM, ecc. Inoltre ci sono anche SFR “di sistema” come lo STATUS, gli FSR, ecc.



- Ogni SFR è indicato da una label predefinita.
- **Il compilatore XC8 definisce queste label in lettere maiuscole.**

Troviamo informazioni su queste label nei files *nomeprocessore.h* e *nomeprocessore.inc* che si trovano nella cartella `C:\Programmi\Microchip\XC8\versione\include\`.

A differenza di quello che capita con Baseline e Midrange, le label degli Enhanced sono piuttosto uniformi, il che riduce drasticamente la necessità di una continua consultazione dei manuali.

Gli SFR sono tutti registri a 8 bit, dato che sono posti nell'area RAM; ognuno di essi, singolarmente o a gruppi, ha una determinata funzione. Il foglio dati del componente specifica questi particolari, ma un aiuto viene dal fatto che in generale tutti questi registri sono pensati seguendo una filosofia di progetto unitaria. Anche passando da un chip ad un altro le funzioni principali sono analoghe, se non identiche.

Microchip definisce come periferiche tutti quei moduli che eseguono una determinata funzione. Sono periferiche i port, i timer, i moduli di comunicazione, i moduli analogici, ecc.

Alcune periferiche richiedono anche più di un SFR per la loro gestione, perchè la loro complessità e le opzioni disponibili richiedono numerosi bit di controllo.

Ad esempio, il modulo di comunicazione seriale EUSART dispone di 16 registri, di cui 6 dedicati (foglio dati di PIC12F1572, **TABLE 21-1**, pag.179), mentre basta un solo FSR per gestire il modulo FVR (Fixed Voltage Reference) che genera una tensione interna di riferimento (foglio dati di PIC12F1572, **TABLE 13-2**, pag.125).

Per contro, non in tutti gli SFR gli bit sono interamente implementati, generalmente perchè occorrono meno di 8 bit per controllare quella data funzione o, come nel caso degli I/O, ci sono port con meno di 8 bit. Ad esempio, nei PIC con 8 pin, i bit utilizzabili nell'I/O sono solo 6 (due degli 8 pin sono impegnati nella tensione positiva e negativa dell'alimentazione), dunque il port è composto da soli 6 elementi che richiederanno solo 6 bit di controllo.

Ecco come è descritto graficamente l'SFR che è identificato dalla label **PORTA** nel foglio dati (**REGISTER 11-1**, pag.110):

REGISTER 11-2: PORTA: PORTA REGISTER							
U-0	U-0	R/W-x/x	R/W-x/x	R-x/x	R/W-x/x	R/W-x/x	R/W-x/x
—	—	RA<5:0>					
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	
u = Bit is unchanged	x = Bit is unknown	U = Unimplemented bit, read as '0'
'1' = Bit is set	'0' = Bit is cleared	-n/n = Value at POR and BOR/Value at all other Resets

Osserviamo che i bit 7 e 6 non sono implementati, dato che non ci sono pin corrispondenti. Scrivendo bit non implementati, il dato va perso e in lettura essi rendono generalmente 0 (in alcuni casi più rari, 1).

Una barra indica il valore che assume ogni bit al POR e le sue caratteristiche di lettura/scrittura: vediamo che tutti i bit implementati sono R/W, cioè leggibili e scrivibili e che dopo un reset il loro contenuto non è definito.

Al POR molti SFR vengono inizializzati con valori specifici (ad esempio, i bit dei registri di direzione TRIS sono posti tutti a 1). Altri SFR contengono valori casuali e se usati, vanno inizializzati dal programma.

Ad esempio è casuale il contenuto di **LATA**, mentre quello di **PORTA** rispecchia le tensioni applicate ai pin.

Ogni SFR corrisponde ad una locazione della RAM ed è, quindi, accessibile attraverso il suo indirizzo nella mappa della memoria (indirizzo assoluto).

Però, un obbligo estremamente importante per una buona programmazione è quello di **non utilizzare mai indirizzi assoluti**. Questo permette la portabilità del programma, dato che lo stesso SFR in PIC diversi potrebbe trovarsi ad indirizzi diversi o su banchi diversi e gli Enhanced dispongono della RAM distribuita su ben 32 banchi.

Se, invece, usiamo esclusivamente le label pre definite nel compilatore, non incorriamo in alcun errore, dato che esso provvede in automatico ad assegnare il giusto indirizzo.

Oltre agli SFR, Microchip definisce anche i **GPR** (General Purpose Register): si tratta delle locazioni di RAM dati, accessibili dal programma in lettura e scrittura, attraverso il loro indirizzo, o meglio, per le quali si dovranno creare label sostitutive.

0.7 - Alcune informazioni di base sul compilatore XC8.

Chi incontra il C per la prima volta si può trovare di fronte ad una situazione dell'informazione molto disordinata, confusa e contraddittoria.

Questo è dovuto a molte cause, ma principalmente dipende dal fatto che **il C non è un linguaggio "univoco", ma si è evoluto in un numero sorprendente di "dialetti" che possono differire tra loro in molti punti.** C è in continua evoluzione e versioni di qualche tempo fa sono superate da altre che possono presentare alcune differenze sensibili, oltre al fatto che l'adattamento ad ambienti particolari, come quello dei microcontroller, richiede considerazioni che sono diverse da quelle dell'ambito grafico o dei sistemi operativi.

A questo si aggiunge la naturale confusione del WEB e la scarsa consistenza di molti contenuti, mentre testi che siano una guida certa per un auto apprendimento non sono certo molti, oltre agli User's Manual dei compilatori, che si presentano sempre di lettura non facile.

Una base unificatrice delle varie versioni è data dalle specifiche ANSI-C e C99 a cui XC8 aderisce. Questa introduzione al compilatore XC8 non è per nulla esaustiva di tutte le possibilità del C, ma cerca di fornire alcuni, pochi, elementi che dovrebbero permettere la comprensione di quanto viene detto nelle esercitazioni seguenti.

Altri elementi saranno aggiunti quando necessario durante le esercitazioni.

0.7.1 - Le parole riservate del compilatore.

Iniziamo col dire che il compilatore XC ha alcune parole che sono riservate.

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>
<code>char</code>	<code>extern</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>float</code>	<code>short</code>	<code>unsigned</code>
<code>continue</code>	<code>for</code>	<code>signed</code>	<code>void</code>
<code>default</code>	<code>goto</code>	<code>sizeof</code>	<code>volatile</code>
<code>do</code>	<code>if</code>	<code>static</code>	<code>while</code>

Alcuni compilatori possono avere un diverso set di parole riservate.

Un uso di queste label al di fuori di quanto imposto dal compilatore genererà errori.

0.7.2 - I formati dei numeri interi in XC8.

Già inizialmente avremo a che fare con la necessità di scrivere numeri e il compilatore XC8 può trattare numeri interi (integer literals) in diversi formati:

- **DECIMALE** (in base 10):
 - sono accettate le cifre da 0 a 9
 - **non possono iniziare con 0**
 - non possono includere virgole, punti o spazi
 - possono essere preceduti dal segno -

Numeri decimali validi	0	5	127	-1024	65535
Non validi	05	5.0	12,7	1.024	6 535

L'uso dei numeri decimali è certamente fondamentale nella matematica, ma la sua estensione a valori di comando per registri di controllo non è certo fonte di sorgenti chiari. Una cosa è attribuire ad un registro ad 8 bit una assegnazione del genere:

```
PORTA = 0b11110000 ;
```

da cui è chiaro che i primi 4 bit alti sono a livello 1, mentre gli altri sono a 0. Meno chiaro è l'equivalente:

```
PORTA = 240 ;
```

dove lo stato dei bit non è certo immediato da individuare.

- **ESADECIMALE** (in base 16)
 - sono accettate le cifre da 0 a 9 e le lettere da A a F
 - **devono iniziare con 0x oppure 0X**
 - non possono includere virgole, punti o spazi
 - possono essere preceduti dal segno -

Numeri esadecimali validi	0x01	0X5	0x1FF	-0X1024	0xFFFF
Non validi	0xG	0X5.0	0X12,7	24h	0x6 5

- **OTTALE** (in base 8). Questo modo è riportato in ANSI-C e può essere impiegato; risale all'epoca di mainframe come i famosi PDP, ma praticamente ora è pochissimo usato.
 - sono accettate le cifre da 0 a 7
 - **devono iniziare con 0**
 - non possono includere virgole, punti o spazi
 - possono essere preceduti dal segno -

Numeri ottali validi	01	05	01024	-01024
Non validi	08	05.0	012,7	029

- **BINARIO** (in base 2). ANSI-C non specifica un formato per i numeri binari, ma i compilatori riconoscono le forme che rispettano le regole seguenti:
 - sono accettate le cifre 0 e 1
 - **devono iniziare con 0b oppure 0B**
 - non possono includere virgole, punti o spazi
 - possono essere preceduti dal segno -

Numeri binari validi	0b1	0B1011	0b11000011
Non validi	0b2	01011	10100b

La notazione binaria è quella che rende graficamente evidente la situazione a livello di bit. Ad esempio, dal numero associato ad un registro:

PORTA = 0b10101010 ;

noto immediatamente che i bit, alternativamente, sono a 1 o a 0.

Se, però, c'è l'obbligo di scrivere un sorgente strettamente ANSI, si dovrà evitare.

La mancanza della notazione binaria sorprende in quanto i “numeri” nelle locazioni di memoria sono presenti nella forma di livelli di tensione 1 e 0 e quindi il binario è nella natura stessa dei computer. Assieme all'esadecimale è la forma più pratica per trattare bit.

All'atto pratico, si utilizzerà la notazione che rende più chiaro quanto si sta eseguendo nel sorgente.

0.7.3 - I tipi di dati del compilatore XC8 e le variabili.

Le variabili sono dati che corrispondono, in generale, a locazioni memoria RAM (volatile) e sono dichiarate per supportare le varie azioni richieste dal programma.

Il bus dati dei microcontroller considerati è a **8 bit**, mentre può essere richiesto di utilizzare **numeri che richiedono più bytes** per essere elaborati ed espressi.

Di conseguenza, quando si devono trattare dati e variabili, è **importante definirne correttamente il tipo**. Da questo dipendono molte cose, come la quantità di memoria che sarà necessaria ad ospitare e trattare il dato e la complicazione delle operazioni su questi dati, quindi la velocità di esecuzione del codice e le sue dimensioni.

Per quanto riguarda i numeri interi, XC8 supporta questi tipi:

Tipo	Dimensione [bit]	Estensione
<code>__bit *</code>	1	0-1
<code>signed char</code>	8	-128 / +127
<code>char</code> <code>unsigned char</code>	8	0 / 255
<code>short</code> <code>signed short</code>	16	-32768 / +32767
<code>unsigned short</code>	16	0 / 65535
<code>int</code> <code>signed int</code>	16	-32768 / +32767
<code>unsigned int</code>	16	0 / 65535
<code>short long *</code> <code>signed short long *</code>	24	-8388608 / +8388607
<code>unsigned short long *</code>	24	0 / 16777215
<code>long</code> <code>signed long</code>	32	$-2^{31} / +2^{31}-1$
<code>unsigned long</code>	32	$0 / 2^{32}-1$
<code>signed long long</code>	32	$-2^{63} / +2^{63}-1$
<code>unsigned long long</code>	32	$0 / +2^{64}-1$

* questi tipi non sono identificati nello standard C99.

La variabile `__bit` occupa un solo bit e il compilatore può accorpere più variabili di questo tipo in un byte. Questo può migliorare l'ottimizzazione del codice.

Data la sua natura, non ha senso attribuire un qualificatore a questo tipo.

Le altre variabili, invece, hanno una radice (in blu) a cui si associa un qualificatore (in nero).



Se si omette il qualificatore, C considera il tipo come `signed`.

Così, scrivendo `int` è come se avessimo scritto `signed int` e ne risulta che il valore massimo (positivo) contenuto non sarà 65535, ma 32768.

Se dobbiamo trattare interi positivi fino a 65535, dovremo dichiarare esplicitamente `unsigned int`.

Sarebbe bello se la regola fosse generale. Purtroppo:



Attenzione, perchè se tutti i tipi non qualificati sono intesi come `signed`, per XC8 `char` è `unsigned`.

`char` è la variabile più piccola a 8 bit, ovvero l'ampiezza del bus dati e delle celle dell'area RAM (SFR, memoria dati) e dal quale, istintivamente, ci si aspetta che copra valori da 0x00 a 0xFF (255 dec).

Questa è semplificazione che permette di evitare il qualificatore viene incontro a questa tendenza, ma, da un altro punto di vista, è una complicazione: per altri compilatori, compresi XC16 e XC30, il tipo `char` non qualificato è `unsigned`, come gli altri tipi.

Questo può creare confusioni ed errori difficili da gestire, soprattutto se si porta il sorgente in altri compilatori.



Per avere chiaro cosa si sta facendo, è assolutamente consigliabile indicare sempre il qualificatore.

In particolare, se abbiamo scelto di compilare secondo le specifiche CCI, l'indicazione del qualificatore è obbligatoria.

In tal senso può essere più chiaro l'uso delle variabili conformi al C99, che vengono incluse nella compilazione richiamando :

```
#include <stdint.h>
```

Saranno disponibili allora dei tipi in forma più compatta:

```
uint8_t x;    // x è unsigned int a 8 bit
int8_t x;    // x è signed int a 8 bit
uint16_t x;  // x è unsigned int a 16 bit
int16_t x;   // x è signed int a 16 bit
uint32_t x;  // x è unsigned int a 32
int32_t x;   // x è signed int a 32 bit
uint64_t x;  // x è unsigned int a 64 bit
int64_t x;   // x è signed int a 64 bit
```

u indica unsigned. Se manca, si intende signed. **_t** vuol dire type. Compatto, certo, ma non immediato...

A proposito di shortcut e della tendenza del C a minimizzare il testo sorgente, va ricordato anche che:



XC8 intende il tipo **int** come principale e ne consente l'omissione.

Ne risulta che **signed numA** è equivalente a **signed int numA** ma anche a **int numA** è una scrittura corretta, mentre **numA** avrebbe lo stesso senso (anche se non certo chiaro per chi non conosca a fondo il linguaggio specifico)

La sintassi è generale è:

```
type nomeVariabile;
```

dove type è il tipo di dato visto sopra. Ad esempio

```
unsigned char characterX;
unsigned int  interA;
int          signd99;
```

Il nome della variabile può contenere caratteri alfanumerici, ma non può iniziare con un numero. Sarebbe possibile utilizzare anche il carattere **_** (*underscore*), ma non se ne consiglia l'uso in quanto viene impiegato nelle librerie.

I tipi visti sopra sono relativi a numeri interi, con o senza segno. XC8 definisce anche tipi per matematica in virgole mobile. Questi tipi comprendono una base, un esponente ed una mantissa.

Tipo	Dimensione [bit]	Minimo assoluto	Massimo assoluto
float	32	$\pm \sim 10^{-44.85}$	$\pm \sim 10^{-38.53}$
double	32		
long double	64	$\pm \sim 10^{-323.3}$	$\pm \sim 10^{-308.3}$

Non tutti i valori tra massimo e minimo sono possibili e il risultato di una operazione in floating point è quanto possibile vicino al reale.

Tutti questi tipi sono a 32 bit e devono essere usate solo dove necessarie, in quanto si trasformano in molte centinaia di istruzioni del linguaggio macchina, con il relativo impegno di memoria e tempo di esecuzione. In questo senso il formato può essere limitato a 24 bit modificando le opzioni del linker di XC8 relative al progetto (da osservare che CCI prevede solo 32 bit).



Una variabile va definita PRIMA di essere utilizzata.

All'atto pratico, dichiarare la variabile equivale ad attribuire una label ad una o più locazioni di RAM. Il contenuto della Ram all'accensione è casuale, quindi la dichiarazione della variabile

```
char j; // dichiarazione di una variabile non inizializzata
```

lascia il valore iniziale di j alla casualità. Dove questo non è accettabile, la variabile può essere inizializzata:

```
char j; // dichiarazione di una variabile non inizializzata
j = 0; // inizializzazione
```

e più brevemente con:

```
char j = 0; // dichiarazione di una variabile inizializzata
```

La riga indica al compilatore di creare una variabile j senza segno, ampia 8 bit (ovvero una locazione di RAM) e caricare 0 in questa locazione.

La riga qui sopra è chiamata anche “istruzione” e che **una riga di istruzioni va terminata con un punto-e-virgola** (semicolon).

Va ben considerato che con “istruzione” non si intendono gli opcodes del linguaggio macchina, ma di comandi che sottintendono anche molti opcodes.

Ad esempio, quanto sopra, in Assembly, richiede almeno 5 linee e due opcodes:

```
cblock 0x20
j
endc
...
movlw 0
movwf j
```

Il compilatore invia un **Warning** se una variabile dichiarata non viene inizializzata.

Siccome è opportuno realizzare sorgenti senza warning, si consiglia di inizializzare sempre le variabili, anche se in pratica il preprocessore provvede a iniziarle a 0.

Variabili composte da più elementi si definiscono **array**. Vedremo qualcosa di più in proposito durante le esercitazioni.

0.7.4 - Le costanti.

Come dice il nome, si tratta di “variabili” che mantengono il loro valore per tutta la durata del programma. Sono allocate nell'area della memoria programma.

Possono essere numeri, caratteri o stringhe.

Sono dichiarate con `#define`.

A esempio:

```
#define size = 16
#define MAX 255
```

Possiamo usare una costante come “label” per un bit di comando o per sostituire un testo:

```
#define LED1 LATAbits.LATA5 // LED! su RA5
#define PI 3.14159
```



Da notare che la riga di `#define` non deve essere terminata con il `;` e che solitamente le label delle costanti sono scritte in lettere maiuscole.

Se utilizziamo `const` per definire la costante, tipicamente indirizziamo questa a trovarsi nell'area della memoria programma.

```
const char [] = "Microcontroller"
```

posiziona la stringa di caratteri in Flash.

0.7.5 - Gli operatori matematici.

XC8 supporta i vari **operatori aritmetici** classici

Operatore	Azione	Esempio	Risultato
*	moltiplicazione	a * b	Prodotto di a e b
/	divisione	a / b	Quoziente di a e b
%	modulo	a % b	Resto della divisione tra a e b
+	addizione	a + b	Somma di a e b
-	sottrazione	a - b	Differenza tra a e b
+	positivo	+ a	Valore positivo di a
-	negativo	- a	Valore negativo di a

In una espressione la priorità dell'operazione sarà quella comunemente adottata nella matematica. Ad esempio:

a - b * c è valutata come **a - (b * c)**
a / b % c è valutata come **(a / b) % c**

Il tipo del dato influisce sul risultato. Ad esempio, se entrambi gli operandi sono **int**, sarà **int** anche il risultato. Ad esempio:

```
int a = 10;
int b = 4 ;
float c;
c = a / b;
```

rende in ogni caso un risultato c di tipo **int**. Per avere un risultato **float** occorre che almeno una delle variabili sia **float**:

```
int a = 10;
float b = 4.0f ;
float c;
c = a / b;
```

Esiste anche una classe di **operatori di incremento e decremento**, molto usati:

Operatore	Azione	Esempio	Risultato
++	incremento	a++	Usa a , quindi incrementalo di 1
		++a	Incrementa a di 1, quindi usalo
--	decremento	a--	Usa a , quindi decrementalo di 1
		--a	Decrementa a di 1, quindi usalo

Ad esempio:

```
i = 0; i++;      // assegna 0 alla variabile i e incrementala di 1
a = 5;
b = (a++) + 5;  // b = 10 e a = 6
```

e:

```
a = 5;
b = (++a) + 5;  // b = 11 e a = 6
```

Esistono, poi, **operatori composti (compound)**, basati sul segno = (assegnazione):

Operatore	Azione	Esempio	Risultato
=	assegnazione	a = b	Assegna ad a il valore di b
+=	Assegnazioni composte	a += b	a = a + b
-=		a -= b	a = a - b
*=		a *= b	a = a * b
/=		a /= b	a = a / b
%=		a %= b	a = a % b
&=		a &= b	a = a & b
^=		a ^= b	a = a ^ b
=		a = b	a = a b
<<=		a <<= b	a = a << b
>>=		a >>= b	a = a >> b

All'atto pratico, **a += b** è lo stesso che scrivere **a = a + b**.

La forma compound, peraltro poco intuitiva, ottiene una riduzione del numero di caratteri necessari all'istruzione. Ad esempio:

```
bVariable &= 0x30;
```

effettua l'AND del contenuto di **bVariable** con **0x30** e salva il risultato in **bVariable** equivalendo a:

```
bVariable = bVariable & 0x30;
```

Attenzione: ripetiamo che una riga di istruzione del C può corrispondere anche a decine o centinaia di istruzioni del linguaggio macchina. Quindi, una valutazione della grandezza del file eseguibile e del tempo di esecuzione è ben difficilmente derivabile dal sorgente C. Occorrerà consultare il listato **.asm** ottenuto.

Le ottimizzazioni operano nella direzione di ottenere liste assembly quanto più possibile ridotte (e tempi di esecuzione corrispondentemente accorciati). Dato che queste operazioni richiedono complessità notevoli nel compilatore, ecco che XC è disponibile "aggratissimo" nella versione base,

minimamente ottimizzata, mentre le versioni via via più ottimizzate hanno un costo. La prima, che qui usiamo, è più che adeguata per l'istruzione e per prodotti finali non critici, mentre le altre sono indirizzate all'attività professionale.

Alcuni segni visti nella tabella precedente sono i classici **operatori bitwise**, comuni ad altri linguaggi.

Operatore	Funzione bitwise	Esempio	Risultato (per ogni bit)
&	AND	a & b	1 se a e b a 1, altrimenti 0
 	OR	a b	1 se a o b a 1, altrimenti 0
^	XOR	a ^ b	1 se a e b sono diversi, altrimenti 0
~	NOT	~ b	1 se b = 0 e viceversa (complemento a 1)

Gli operatori bitwise eseguono operazioni booleane bit per bit sulle variabili indicate.

E' indispensabile non confondere gli operatori bitwise con gli **operatori logici**, che eseguono operazioni booleane sull'intera variabile:

Operatore	Funzione logica	Esempio	Risultato
&&	AND	a && b	1 se entrambi a e b sono diversi da 0
 	OR	a b	0 se entrambi a e b sono 0
!	NOT	!a	1 se a = 0

Per maggiore chiarezza:

0b1010 & 0b1101 rende **0b1000**

dato che si effettua una operazione di AND bit per bit.

```

0b1010 &
0b1101
-----
0b1000      AND = 1 solo se entrambi i bit sono a 1

```

Per contro

0b1010 && 0b1101 rende **0b0001** (cioè VERO)

perchè si tratta di AND logico, che considera non i singoli bit, ma tutto il valore. Qui, entrambi gli elementi non sono a 0 per cui il risultato sarà 1.

Altro esempio, la sequenza:

```
char a = 0b1010;
char b = 0b0101;
if (a & b) printf("TRUE")
```

non stamperà **TRUE** perchè l'AND bitwise rende **0b0000**
Invece:

```
char a = 0b1010;
char b = 0b0101;
if (a && b) printf("TRUE")
```

stamperà **TRUE** perchè l'AND logico tra i due elementi rende **0b0001**, dato che entrambi sono diversi da 0.

Analogamente, troviamo degli **operatori relazionali aritmetici** :

Operatore	Funzione	Esempio	Risultato (VERO=1, FALSO=0)
<	minore di	a < b	1 se a minore di b , altrimenti 0
<=	minore o uguale	a <= b	1 se a minore o uguale a b , altrimenti 0
>	maggiore	a > b	1 se a maggiore di b , altrimenti 0
>=	maggiore o uguale	a >= b	1 se a maggiore o uguale a b , altrimenti 0
==	uguale	a == b	1 se a uguale a b , altrimenti 0
!=	non uguale	a != b	1 se a diverso da b , altrimenti 0

Espressioni che rendono valori diversi da 0 sono interpretate come VERO. Se il risultato è 0 è in qualsiasi caso FALSO.



Da notare che il segno **==** non ha la stessa funzione di **=**:

- **=** indica una **assegnazione**. Con **a = b** si intende: rendi **a** uguale a **b**
- **==** indica una **relazione**. Con **a == b** si intende: se **a** è uguale a **b**

Scrivendo **x = 5** si assegna il valore **5** alla variabile **x**.
Volendo imporre una comparazione si scriverà:

```
if (x == 5) {
    //esegui questo se x è uguale a 5
}
```

In alcuni algoritmi è pratico disporre di **operatori di shift**:

Operatore	Funzione	Esempio	Risultato
<<	shift left	a << b	shift a di b volte a sinistra
>>	shift right	a >> b	shift a di b volte a destra

Ad esempio:

```
a = 6           // a = 0b00000110
x = a << 2;     // x = 0b00011000 = 24
```

Se si applica lo shift ad un numero con segno, questo viene conservato:

```
a = -6         // a = 0b11111010
x = a >> 2;     // x = 0b11111110 = -2
```

Da ricordare che uno shift a sinistra equivale ad una divisione per 2, mentre uno shift a destra equivale ad una moltiplicazione per 2.

```
x = a / 4      equivale a   x = a >> 2
x = a * 2      equivale a   x = a << 1
```

L'uso degli shift rende più compatto il codice ottenuto dalla compilazione.

Esistono, poi, numerosi altri operatori e forme di scrittura che, dove necessario, vedremo durante le esercitazioni.

In ogni caso, è opportuno consultare il manuale utente del compilatore nella versione usata.

0.7.6 - I segni speciali.

Una nota:



Purtroppo nel layout della tastiera italiana le parentesi graffe `{ }` sono solitamente assenti. Si potranno richiamare con:

- con le sequenze `Alt`.
`Alt+123` genera `{` e `Alt+125` genera `}`. I numeri devo essere battuti sul tastierino numerico. Su un notebook potrà essere necessario `Alt+Fun`
- la sequenza `AltGr+shift+[` per il segno `{` e `AltGr+shift+]` per il segno `}` è utile dove manca un tastierino numerico, come in molti notebook

Il simbolo `~` (tilde) si ottiene con la sequenza `Alt+126` o `Alt+Fun+126`.

In tutti i casi si può ricorrere alla *Mappa Caratteri* di Windows.

Può essere utile disporre di una tastiera con layout USA, che evidenzia questi segni speciali.

0.8 - Alcuni informazioni sul compilatore XC8.

La compilazione di un sorgente C passa attraverso fasi abbastanza complesse. Per avere una idea di cosa accade durante la compilazione, vediamo nello schema seguente il suo flusso:

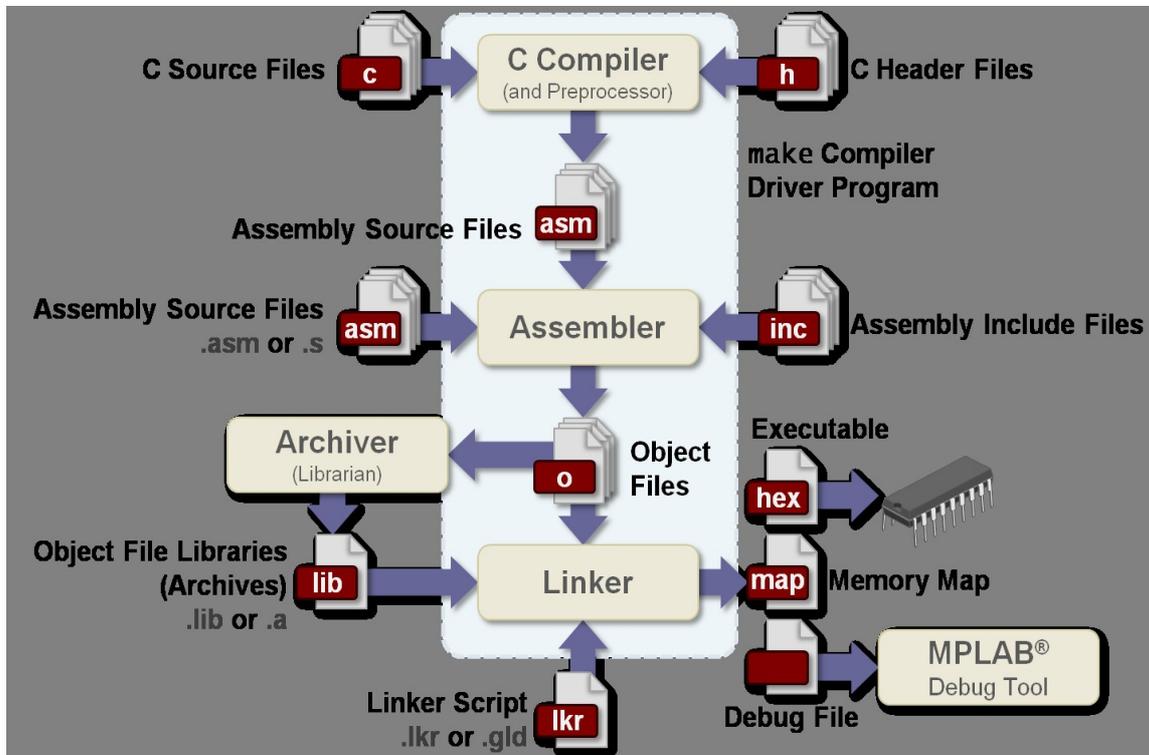


Immagine dal sito di Microchip

Il file sorgente C viene trattato da un preprocessore assieme ai file headers inclusi.

Il compilatore in se comprende tre elementi:

- **Preprocessore** : il suo compito principale è quello di eliminare tutti gli elementi che sono introdotti dal programmatore per rendere il sorgente leggibile: i commenti e gli spazi bianchi inutili vengono rimossi.
Le etichette di sostituzione del testo e le macro sono sostituite dai loro valori reali e il contenuto del file di intestazione viene fuso nel file sorgente C.
- **Parser**: riceve l'elaborazione del pre processore ed esegue la maggior parte del lavoro nel compilatore.
 - Analisi lessicale: Crea i token (caratteri o stringhe che indicano elementi significativi, come parole chiave, operatori matematici o nomi di variabili).
 - Analisi Sintattica: Si assicura che i token formino espressioni conformi alle regole di C.
 - Analisi semantica: Determina quali azioni devono essere intraprese da ogni espressione e passa una lista di queste azioni al generatore di codice.
- **Generatore di codice** : è unico per ogni architettura di microcontroller, dato che deve prendere una lista di azioni che il programma deve eseguire dal Parser e tradurle in istruzioni specifiche del linguaggio Assembly del dispositivo. Il codice Assembly generato è riposizionabile, il che significa che nulla in esso è stato assegnato ad un indirizzo fisico sul

dispositivo. Determinare dove il codice e le variabili risiederanno nello spazio di memoria è il compito del Linker.

Questa sequenza origina un file Assembly, che viene passato all'Assembler assieme ad altri file *.asm* o *.inc* dove richiesto. Il passaggio attraverso il Linker produrrà i file finali di mappa ed eseguibile, assieme alla possibilità di effettuare un debug.

Il compito del **Linker** è quello di combinare insieme file oggetto e file di libreria (essi stessi, semplicemente una collezione di file oggetto) in un unico file eseguibile o, nel caso dei microcontrollori, un file *.hex* o 'file esadecimale'.

I file oggetto sono riposizionabili, il che significa che il codice e i dati in essi contenuti possono essere posizionati in qualsiasi punto della mappa di memoria del dispositivo. In altre parole, non ci sono indirizzi hard-coded nel sorgente originale, salvo alcune eccezioni. Questo permette di mescolare file sorgente e librerie per lo stesso microcontroller in un singolo progetto. Non ci possono essere conflitti di indirizzo se le variabili e i blocchi di codice non vengono forzati ad un indirizzo specifico nella mappa della memoria del dispositivo.

Quindi, il compito principale del linker è quello di capire dove tutto il codice e i dati risiederanno nella memoria del microcontroller. Assistente in questo compito è lo script linker che definisce la struttura della memoria del microcontroller e quali posizioni possono o meno essere assegnate dal linker per uno scopo specifico. Con queste informazioni, il linker tenta di localizzare ogni blocco di codice e ogni variabile ad un indirizzo specifico nella mappa della memoria del dispositivo.

A questo punto, il programma è nella sua forma finale e il linker emette un file *.hex* che contiene l'immagine binaria da programmare nella memoria flash del dispositivo. Se il progetto è stato compilato per il debug, il linker produrrà invece un file *.elf* (o un file *.coff* o *.cod* per compilatori e IDE più vecchi) che contiene la stessa immagine binaria del file *.hex*, ma con uno speciale codice di debug che è necessario ai tools per interagire con la periferica.

[Altre informazioni sul file *.hex* qui](#).^(www)

Infine, il linker può opzionalmente produrre un file *.map* che mostrerà dove ogni variabile e blocco di codice è stato organizzato nello spazio di memoria del dispositivo.

Come si vede, si tratta di una catena molto complessa.

Complessità che viene ancora aumentata quando si aggiungono le funzioni di ottimizzazione.

In ogni caso, sui PC attuali, le operazioni di compilazione impiegano frazioni di secondo e solo davanti a compilazioni molto estese e complesse si notano attese di qualche secondo.

0.9 - Alcune domande comuni.

Quanto tempo è necessario per imparare il C?

Questo dipende dalle vostre capacità e dal tempo dedicato. In generale si parla di un tempo variabile tra 1 e 6 mesi. C non è un linguaggio facile, soprattutto per la mancanza di forma e per la grande quantità di shortcut e trucchi possibili. La presenza di una molteplicità di dialetti, poi, fa sì che il passare da uno all'altro non sia così immediato.

Inoltre, l'operazione di creare un programma e trasferirlo in un chip richiede, contemporaneamente, anche la conoscenza dell'ambiente di sviluppo, che, di per se, richiede ulteriore impegno.

Da considerare anche che il realizzare una applicazione embedded ha bisogno, oltre alla conoscenza del linguaggio e dell'IDE di sviluppo, anche adeguate conoscenze delle particolarità dei processori con cui si vuole lavorare.

Occorre avere una conoscenza di base tanto del linguaggio quanto dell'elettronica, altrimenti ci si fa poco.

Che differenza c'è tra C e XC8?

XC8 è un compilatore per i processori di Microchip. Pur essendo un ANSI C, è indirizzato a trattare applicazioni in un ambito molto diverso da quello per cui sono previsti C++, C#, Visual C e simili.

Che differenza c'è tra XC e Arduino (o tra PIC e Arduino)?

La domanda è un non senso che nasconde una notevole mancanza di conoscenze.

XC è una versione del linguaggio di programmazione C.

Arduino è una scheda di prototipazione, che usualmente impiega una versione di C per i suoi programmi. I processori usati sulle schede base di Arduino possono essere meno performanti delle più recenti versioni di PIC, ma il vantaggio sta nella disponibilità di un elevato numero di librerie che coprono molteplici applicazioni; questo, col metodo del taglia-e-incolla, consente anche a utenti abbastanza sprovveduti di arrivare a realizzare questo o quel circuito.

Usando il C ANSI è possibile accedere a tutte le risorse hardware di un micro esattamente come si fa in assembly e con la stessa efficienza...

Assolutamente no, se con questo si intende che la conoscenza, anche approfondita, di un C genericamente ANSI consenta di programmare senza problemi qualsiasi genere di microcontroller. Ogni famiglia di MCU ha caratteristiche proprie che possono differire sensibilmente da altre famiglie; questo è vero anche all'interno dei prodotti di un unico costruttore. Tanto più è differente il prodotto di un diverso costruttore che progetta seconda una differente filosofia.

Per potere usare moduli integrati (UART, I2C, CAN, USB, timer, PWM, ADC, ma anche i semplici I/O digitali) occorre conoscere le caratteristiche del singolo dispositivo su cui si sta lavorando, quali registri vanno maneggiati e come, ecc. Senza conoscenza di come funziona l'hardware e di quali comandi si devono dare, in ambiente embedded, se non ci sono librerie specifiche, non si combina nulla.

In particolare, il C, pur essendo dichiarato come il linguaggio più vicino all'Assembly, non è l'Assembly e non ne ha la stessa efficienza a livello di codice generato. Tanto che C consente con facilità di integrare linee di Assembly proprio per parare quelle situazioni dove il linguaggio non potrebbe arrivare. Per contro, C dispone della possibilità di realizzare in poche linee blocchi logici che in Assembly ne richiederebbero molte decine o centinaia.

Quali sono i vantaggi dell'uso del C?

- **Efficienza** – permette di scrivere sorgenti molto più brevi che in Assembly
- **Applicazioni** – l'uso del C si è esteso ad una grandissima quantità di ambiti, dalla creazione di sistemi operativi ai microcontroller embedded. La conoscenza delle basi del C è applicabile ampiamente.
- **Documentazione** - sono disponibili corsi, tutorial, documentazioni ed esempi in grande quantità. Esistono anche buoni libri.
- **Embedded** – pur essendoci BASIC, Fortran ed altri, C è l'unico linguaggio a livello più alto dell'Assembly che si è diffuso ampiamente nell'ambiente embedded professionale. La maggiore integrazione con l'hardware lo fa definire come il linguaggio più basso di quelli ad alto livello.

Quali sono gli svantaggi del C?

- **Non portabilità** – un sorgente C scritto per una certa architettura non è portabile su una differente.
Le funzioni dello standard ANSI sono portabili, ma non lo sono le librerie e i codici specifici di un determinato genere di processori e delle loro periferiche integrate. Non appena si fa un uso intensivo di librerie non ANSI, ci si ritrova bloccati a quel dato compilatore e in difficoltà a portare i sorgenti.
- **Non è univoco** – esiste una miriade di dialetti e variazioni, anche attorno allo standard base ANSI C, soprattutto dove il linguaggio è applicato ad ambienti specifici. Può non essere immediato (o possibile) passare un sorgente da un dialetto ad un altro.
- **Non è semplice da apprendere** – le sue caratteristiche lo rendono di non immediata comprensione ed è richiesto un certo tempo per adattarsi alla sua logica.
- **Efficienza** - non è efficiente come Assembly: per quanto sia ottimizzato, il codice ottenuto dalla compilazione userà comunque più memoria e, in generale, produrrà codici meno compatti. In particolare, C è pensato per trattare array, matrici multi dimensionali, masse di dati da convertire e stockare, mentre non è stato pensato per manipolare i singoli bit, attività preminente nei microcontroller. I dati, in questo ambito, vengono “consumati” subito per trasformarli in azioni su quanto collegato al micro. Il fatto che tipi per un singolo bit o l'uso di numeri binari esistano solo come aggiunte allo standard, dovrebbe essere sufficiente a comprendere la situazione.
- **Non è minimamente auto documentante** – anzi, C permette la scrittura di testi confusi (obfuscated) con gran facilità. La mancanza di una struttura del testo (indentation) e di seri commenti può rendere il sorgente illeggibile anche a chi lo ha scritto.
- **Documentazione poco efficace** - C'è certamente una gran massa di informazione sul web, ma spesso tremendamente confusa dalla pletora di variazioni del linguaggio, e, come tipico di Internet, superficiale, poco approfondita e in gran parte copia della copia di copie. Alla fine, districarsi e selezionare quello che serve realmente non è semplice. Anche per quanto riguarda i libri occorre una seria selezione.

E' meglio C o BASIC?

Sono due cose differenti, ma se ci rivolgiamo all'ambiente embedded, C è maggiormente diffuso in campo professionale per le migliori prestazioni e per la possibilità di una integrazione maggiore con

l'hardware che deve comandare. Microchip ha realizzato compilatori C e non BASIC.

Per i microcontroller c'è solo XC?

No. Esistono svariati altri compilatori, da quelli supportati da noti produttori (come ad es. Mikroelektronika) a quelli open source, come SDCC, facilmente scaricabili da internet.

In queste esercitazioni usiamo XC perchè è il compilatore ufficiale del produttore dei PIC e degli AVR.

E' utile avere una licenza di XC?

Se non si tratta di lavoro, no. La versione free non ha alcuna limitazione rispetto a quelle a pagamento; manca solamente delle ottimizzazioni, che, a livello hobbistico o di studio, non hanno particolare utilità. La cosa sarà da considerare in modo differente se si utilizza il compilatore in ambito professionale.

Peraltro, la versione free di XC8 dispone della possibilità di attivare per un tempo limitato una licenza con ottimizzazione, in modo da permettere all'utente di valutare l'utilità dell'acquisto.

Quanto detto sopra non esaurisce certamente la conoscenza del microcontroller e tanto meno del C, ma si tratta solo degli elementi minimi necessari per poter iniziare le esercitazioni.

Altre informazioni saranno fornite quando necessario durante le esercitazioni

Se siete sopravvissuti a questa introduzione, potete passare alla prima esercitazione.

0.10-Alcuni link utili.(www)

- [Fundamentals of the C Programming Language](#) da Microchip
- [Microchip Developer Help](#) da microchip
- [MPLAB XC8 compiler](#) da Microchip
- [Software development tool](#) da Microchip
- [Self Paced Training](#) da Microchip