crestus

TUTORIAL PIC - LA PROGRAMMAZIONE

9 November 2010

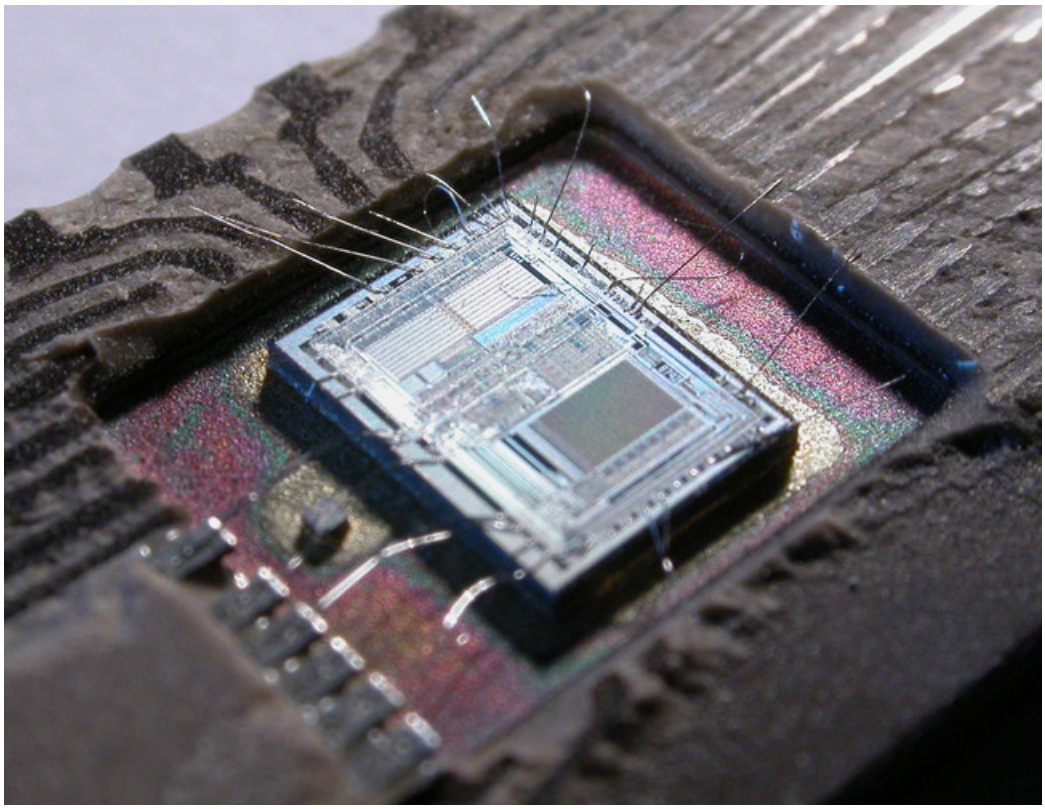
I linguaggi di programmazione

Ora che avete scelto quale modello utilizzare e che sapete tutto quello che c'è da sapere per farlo funzionare bisogna creare, però, il **programma**!

I microcontrollori hanno bisogno di un tipo particolare di **istruzioni**, perché non sono in grado di interpretare concetti che per noi sono molto semplici! Non possiamo scrivere un programma come se fosse un testo!

La logica di un microprocessore è molto limitata, sono "stupidi" di per sé; siamo noi che dobbiamo farli diventare intelligenti e farli reagire come vogliamo noi! Un pensiero che potrebbe venire in mente è il seguente: " Ma se sono stupidi allora anche il loro linguaggio sarà stupido e quindi semplice!".

Beh... in parte questo è vero. le istruzioni che il processore può capire sono spesso limitate, e sono direttamente dipendenti dalle **operazioni** che è in grado di compiere.



CHIP

In realtà il microcprocessore può comprendere solo un linguaggio per noi completamente illeggibile chiamato **“linguaggio macchina”**.

Questo è una sequenza di numeri esadecimali organizzati in **“word”** che creano la sequenza di operazioni che produrranno il risultato voluto. Un esempio di come si presenta un file **“.hex”** lo trovate qua sotto.

```
:06000000E4EF01F0120024
:0A0006000100260400006C00000059
:0400100002000000EA
:0C001400D9CFE6FFE1CFD9FF020EE126B4
:10002000FC0EDBCFF3FFDB06F350000938E027ECD2
:1000300001F0F3CFDEFFF4CFDDFF010EDB50FF0B4D
:1000400003E0FF0E2ED00BD0FD0EDBCFE9FFDB2A45
:10005000FE0EDBCFEAFF01E3DB2ADFCFEFFFFC0E72
:10006000DB500EE194969488010EE66EFFEC01F0F1
:10007000E5529486010EE66EFFEC01F0E5520ED0DB
:1000800094968B989498010EE66EFFEC01F0E55281
:100090009486010EE66EFFEC01F0E552C1D7000E2A
:1000A00000D0026E020EE15C02E2E16AE552E16E0E
```

```
:0A00B0000250E552E7CFD9FF12001D
:0600BA00060EF66E000EBA

:02038000120069
:0E0382008B9694969488010EE66EFFEC01F0C7
:10039000E5529486000000082A803D0FF0E03D02F
:0803A00002D0000E00D0120093
:0803A800000EF36E00EE00F000
:1003B000030E01D81200EA6002D0EE6AFCD7F350B7
:0803C000E9601200EE6AFCD7AF
:0803C80011EE00F021EE00F03F
:1003D000F86A019C5DEC00F012EC02F0EAEC00F02F
:0403E000FBD7120035
:0C03E400030EE66EFFEC01F0E55282B65D
:0E03F00003D0FE0E03D002D0000E00D012008B
:0203FE00FF0EF0
:10040000E350E84E1200000000D000D000D0E82EEB
:04041000FAD7120005
:0C0414008B989498010EE66EFFEC01F04E
:04042000E55212008F
:020424001200C4
:0204260046008E
:020000040030CA
:0100010022DC
:010003001EDE
:0100060001F8
:00000001FF
```

Per creare questa sequenza di numeri, era comunque necessario un linguaggio che facesse corrispondere ad ogni operazione un nome comprensibile, che esprimesse l'operazione che sarebbe stata compiuta.

Il linguaggio che ne deriva è il linguaggio **assembler**.

La particolarità di questo linguaggio è quella di scrivere righe di codice che esprimono esattamente le operazioni che esegue il microcontrollore. Tramite questo linguaggio potrete avere accesso a tutte, e dico proprio tutte, le possibilità offerte dal micro.

Perché allora esistono altri linguaggi? Perché fare qualsiasi operazione richiede molte istruzioni, e la conoscenza assoluta delle caratteristiche del microcontrollore. Questa non è una cosa facilmente ottenibile, ma comunque realizzabile. Esiste un altro problema più importante: la perdita della visione d'insieme del programma; se per prendere una decisione sono necessarie 5 istruzioni diverse, è facile perdere la

prospettiva, inoltre è difficile, a volte, riuscire a vedere il vero significato di una certa operazione! Altro problema ancora è che c'è il rischio di perdersi nei registri e nelle allocazioni di memoria!

```
";************************************************
; Pic by example
; LED.ASM
;
; (c) 2001, Sergio Tanzilli
; <a href="http://www.tanzilli.com/">http://www.tanzilli.com</a>
;************************************************

        PROCESSOR       16F84
        RADIX           DEC
        INCLUDE         "P16F84.INC"
        ERRORLEVEL      -302

        ;Setup of PIC configuration flags

        ;XT oscillator
        ;Disable watch dog timer
        ;Enable power up timer
        ;Disable code protect

        __CONFIG        0x3FF1

LED     EQU     0
        ORG     0x0C

Count   RES     2

        ;Reset Vector
        ;Start point at CPU reset

        ORG     0x00
        bsf     STATUS,RP0
        movlw   B'00011111'
        movwf   TRISA
        movlw   B'11111110'
        movwf   TRISB
        bcf     STATUS,RP0
        bsf     PORTB,LED
```

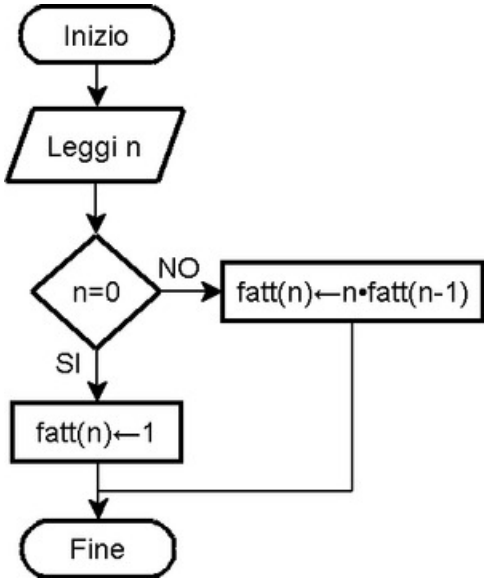
```
MainLoop

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero
        bsf     PORTB,LED
        goto    MainLoop

SetToZero

        bcf     PORTB,LED
        goto    MainLoop
        ;Subroutines
        ;Software delay

Delay
        clrf    Count
        clrf    Count+1

DelayLoop

        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

Conclusione: lasciamo l'assembly ai veri esperti e cerchiamo un altro linguaggio di programmazione che ci faciliti la vita. La soluzione che vi propongo io è il linguaggio C (ANSI C).

In questo articolo non troverete una lezione sul linguaggio C perchè sarebbe impraticabile, ma vi proporrò una guida alle principali istruzioni e alle principali caratteristiche, che vi permetta di capire il ragionamento che sta dietro ad un programma e che vi permetta di leggere con facilità i file sorgenti che incontreremo nel nostro percorso.

Il Programma

Esempio di diagramma di flusso



Programma di esempio

Un programma per microcontrollore, ha come compito generare una sequenza di operazioni, e alcune di queste sono “condizionate”. Per farlo c’è bisogno di un codice che possa essere capito dal compilatore e trasformato in codice assembler. Le operazioni principali che si faranno, saranno legate alla manipolazione di registri e di variabili.

Le variabili possono essere di vario tipo e caratteristiche. Partiamo dall’”unsigned char” lungo 1 byte (8 bit) e con valori compresi tra 0 e 255; passiamo quindi al “signed char” che varia da -128 a +128 (il MSB most significant bit ha valore di segno); poi ci sono gli “int” (interi) che sono lunghi 2 byte e che quindi possono raggiungere il valore di 65535; fino a scomodare i float, che hanno invece una struttura molto più complessa e che permettono rappresentazioni di numeri con la virgola.

I registri invece sono delle parti di memoria che possono essere modificate dal programma e che servono a regolare il funzionamento del micro. Questi sono della lunghezza di 1 byte e attribuiscono un significato specifico ad ogni singolo bit. le operazioni che possiamo fare sono le stesse che possiamo fare sulle variabili, ma conviene andare a trattare i registri in altra maniera, lavorando sui singoli bit.

Normalmente un programma per PC ha un inizio ed una fine. Nell’ambito micro questo procedimento è possibile farlo, ma non è quello che nella quasi totalità dei casi serve fare. Vorremmo che i nostri processori controllassero continuamente il mondo che gli sta attorno, in maniera che sia reattivo nel momento del bisogno in qualunque momento avvenga. Il mezzo che si utilizza è il cosiddetto Loop.

Struttura di base del programma

Un programma per microcontrollori non differisce di molto da un programma per computer: le basi sono esattamente le stesse.

Il programma deve essere contenuto tutto all'interno di un unico blocco definito "main" per definirlo bisogna sempre usare una particolare sequenza che è la seguente, dove al posto dei puntini verrà scritto il programma.

```
void main (void) {

...

}
```

Purtroppo questo non è sufficiente. Qualsiasi istruzione che scriveremo, necessiterà di riferimenti specifici a seconda del microprocessore utilizzato. Inoltre, per poter utilizzare funzionalità avanzate, sarà necessario aggiungere le informazioni delle librerie che ci permetteranno di svolgere determinate operazioni.

Si utilizza allora un costrutto chiamato "include" che va inserito prima del programma principale e che serve a comunicare al compilatore come si chiamano i files dove troverà le informazioni necessarie:

```
#include <p18f14k50.h>
#include <libreria.h>

...

void main(void) {}
```

In questa zona verranno inserite anche tutte le altre informazioni necessarie al compilatore per settare correttamente il PIC e per interpretare correttamente il codice.

I cicli

I cicli sono un sistema che permette al programma di ripetere determinate operazioni più di una volta in sequenza. Potrebbe interessarci eseguire una certa operazione un numero finito di volte, ad esempio accendere e spegnere un led per 3 volte, oppure continuare a fare operazioni fintanto che non si raggiunge la condizione desiderata, oppure all'infinito. Le possibilità sono 2:

- L'istruzione **"for()"** per il numero limitato di volte. Questa istruzione utilizza una variabile che esegue il conteggio delle ripetizioni:

```
for ( i=0 ; i<100 ; i++ ) { }
```

ad esempio, esegue le istruzioni tra le parentesi graffe fintanto che la variabile i, che parte da zero, è minore di 100: 99 + 1 istruzioni, nel caso in cui i = 0, fa proprio 100 cicli), l'ultimo pezzetto è "i++" che sta a significare : "incrementa la i di 1".

- L'istruzione **While()** si preoccupa di eseguire a tempo e numero di volte indefinito una serie di operazioni. Possiamo tradurre in Italiano questa istruzione con un "fintanto che la i è = 0 continua a fare queste operazioni" cioè:

```
while ( i==0 ) {}
```

Come vedete non c'è nessun riferimento al numero di operazioni da compiere, ma c'è un rovescio della medaglia: se la "i" già era diversa da zero non vengono eseguite nemmeno una volta le istruzioni contenute dal ciclo while, che viene completamente ignorato.

Scelte Condizionate

Ci sono momenti in cui il micro si trova a dover scegliere tra 2 o più possibilità, e non è caso infrequente.

Potremmo trovarci a dover scegliere la linea d'azione in base al valore di una variabile: se troppo basso adotteremo una linea d'azione, altrimenti un'altra strategia. Potremmo invece trovarci a lavorare con possibilità multiple: ad esempio indicare il giorno della settimana: abbiamo 7 condizioni differenti tutte dipendenti dalla stessa variabile! questi casi si traducono in 2 possibilità:

- L'istruzione **"if ()"** permette di scegliere tra SI e NO. Esempio:

```
if (i==0) {

 ...

}
else {

 ...

}
```

cioè, se "i" vale zero allora elaboro le istruzioni immediatamente successive e tralascio l'"else", che invece verrà eseguito nel caso in cui la "i" sia diversa da zero.

É possibile scrivere anche condizioni che abbraccino un insieme di valori scrivendo:

```
if (i<2) {...}
```

.

Verrà cioè eseguito il codice tra le parentesi graffe per il valori di “i” compresi tra 0 e 1.

Notate inoltre che non è stata scritta l’istruzione “else{}”; infatti non è necessaria al fine di una corretta sintassi, ma si scrive nel caso ci siano operazioni specifiche da compiere nel caso di condizione insoddisfatta.

- Nel caso di scelta multipla si può utilizzare l’istruzione **“Switch ()”**. Questa istruzione ha una sintassi piuttosto complessa:

```
Switch(var) {

 case 1:
  ...
  break;

 case 2:
  ...
  break;
}
```

Quello che vedete è la struttura di un’istruzione “switch”.

Le parti principali sono:l’inizio: Swtch(i) che introduce la variabile che andrà analizzata; seguono i “case”: in queste istruzioni il valore dopo il "case" indica il valore che dovrà assumere la variabile per eseguire il codice contenuto in quella porzione di istruzione. Il **"break”** alla fine di ogni sezione è necessario!

L’utilità di questa riga è che permette di saltare alla fine dell’istruzione “switch” evitando di compiere altre istruzioni inutili. Questo significa che lo switch è una scelta condizionata esclusiva, cioè ci può essere solo un valore specifico per il quale si ottiene l’esecuzione del codice, non ci possono essere range di valori.

Operatori logici

Cosa succede se invece di lavorare con dei numeri si avesse bisogno di interpretare i singoli bit di una variabile?

Le istruzioni normali non vanno più bene… Bisogna allora usare gli operatori logici! Le istruzioni che permettono queste manipolazioni sono raggruppabili sotto 2

categorie: operazioni di manipolazione del byte con logica bit a bit e operazioni logiche su numeri.

Andiamo prima di tutto a vedere le operazioni sui numeri:

- **=** : descrive una operazione di assegnazione: “var = 12;” significa assegnare il valore 12 alla variabile var.
- **==** : è il confronto invece: “if (var==0) {}” va a verificare se “var” è uguale a 0. Errore comune è quello di dimenticare un uguale all’interno della condizione creando problemi nell’esecuzione e spesso è difficile andare ad individuare il problema…:)
- **!=** : è il suo opposto, controlla che la variabile non abbia il valore indicato
- **<**, **>** : sono ancora istruzioni di confronto del dato: controllano se la variabile a sinistra è minore/maggiore di quella a destra del simbolo
- **<=**,**>=** : sono lo stesso simbolo ma con l’aggiunta dell’uguale e il significato è analogo: se vogliamo includere anche il valore esatto della variabile di destra alla condizione
- **+**, **-**, *, **/** : sono invece le operazioni matematiche somma sottrazione moltiplicazione e divisione.
- **+=**, **-=**, ***=**, **/=** : sono invece operazioni ottimizzate: Se dobbiamo aggiungere alla stessa variabile un’altro numero possiamo scrivere in maniera contratta: “var += 12;” che si potrebbe tradurre con: “var = var + 12;” che fornisce però un codice meno ottimizzato!
- **++**, **-** : sono anche loro delle operazioni ottimizzate, in particolare aumentano o diminuiscono il valore della variabile di 1 sfruttando delle ottimizzazioni molto spinte riducendo anche ad una sola riga di codice: ” var ++;”

Questi sono praticamente i simboli che utilizziamo di solito nella matematica “cartacea” in fin dei conti.

Ma è discorso diverso se invece di operazioni dovessimo fare delle condizioni logiche, come se ad esempio avessimo più di una condizione per poter accedere ad una parte di programma.

Gli operatori che ci permettono di fare Questo sono i seguenti:

- **&&** : è l’operatore logico per l’AND tra condizioni: “if ((var > 0) && (var < 11)) {}” cioè se la variabile var è compresa tra 1 e 10.
- **||** : è il simbolo per l’OR logico: pensate a cosa significa “if ((var < 11) || (var > 20)) {}”….
- **&**, **|** : servono invece se dovessimo andare a manipolare un byte andando a fare le operazioni logiche sui bit modificando il valore del byte stesso! ad esempio: “var &= 0b00001000″ significa: “var = var & 0b00001000″.

sapendo che 0bxx….xx è la rappresentazione di un numero in binario si capisce che andiamo ad isolare il 4° bit della variabile var. In questo caso se quel bit varrà 1 tutto il numero sarà maggiore di 1.

Funzioni

Le funzioni sono utilissime! Spesso le utilizzate senza neanche rendervene conto. La funzione è un sistema per racchiudere una serie di operazioni in un pacchetto, richiamandole con una sola riga di codice. Se avete studiato l'assemble,r possono essere assimilate senza tanti problemi alle MACRO. L'obbiettivo delle funzioni è di rendere leggibile il codice, ma soprattutto di ridurre l'occupazione della memoria. Mettete il caso che nel vostro programma dobbiate eseguire la stessa sequenza di operazioni su un certo numero di variabili in punti diversi del programma. Senza l'ausilio delle funzioni, in ogni punto del programma in cui è richiesta quella sequenza dovrete riscriverla ogni volta, con l'unica modifica del nome della variabile. Inserendo quelle operazioni in una funzione, con riferimento ad una variabile generica, scriverete solo una volta le istruzioni, richiamandole, a mezzo funzione, dove necessario nel programma. Come risultato otterrete una sola riga di codice che effettua le operazioni, rendendo leggibile il listato, e occuperete con quelle operazioni solo lo stretto indispensabile della memoria, anzichè occupare spazio per n volte con le stesse istruzioni.

La struttura sintattica delle funzioni è molto semplice in realtà:

```
risultato = Nome_della_funzione ( variabile1, variabile2, .... );
```

sembra complessa ma non lo è. In pratica passate alla funzione tutte le variabili e i parametri di cui ha bisogno, nella forma richiesta. Come risultato la funzione potrà passarvi una sola variabile. Uniche cose a cui bisogna fare attenzione con le funzioni, sono:

- la sintassi, deve essere chiara in mente
- quando si passano variabili alla funzione che devono essere modificate bisogna passare i loro puntatori

Quest'ultima è legata al fatto che la funzione in realtà lavora con il valore della variabile, ma non con la variabile stessa…possiamo pensarlo come: "si fa una copia della variabile e lavora con quella senza modificare l'originale" anche se non è corretto a rigore affermare questo.

Un esempio tipico delle funzioni è il seguente:

```
// prototipo di funzione
void inizializza_variabili (char var[], int var2, unsigned char var3);
```

```
// la funzione
void inizializza_variabili (char var[], int var2, unsigned char var3) {

 for (i=0; i<10; i++) var [i] =0;
 var2=0;
 var3=0;
}

...

void main (void) {
 inizializza_variabili (vettore, a, temp);

...

}
```

Quì sopra vedete che la funzione viene

1. prima dichiarata per permettere al compilatore di riconoscerla
2. poi viene definita con tutte le sue operazioni
3. e poi, all'interno del main(){ }viene utilizzata.

Se avete osservato attentamente, avrete notato che il ciclo for non ha parentesi graffe. Questo perché se da un'istruzione dipende soltanto un'altra singola istruzione, si possono omettere le parentesi graffe e, come in questo caso, si può anche evitare di andare a capo scrivendo l'istruzione sulla stessa riga.

Istruzioni per il compilatore

Abbiamo parlato più su di istruzioni per il compilatore. Queste istruzioni non sono in realtà legate al linguaggio C, ma proprio al compilatore che abbiamo scelto di utilizzare. Al di sotto degli "#include" possiamo andare a definire alcuni parametri specifici per configurare alcune funzionalità del microcontrollore.

La direttiva che dobbiamo utilizzare è la **"#pragma"**:

```
#pragma config FOSC = HS //Alta frequenza ( 12MHz )
#pragma config WDTEN = OFF //Disabilito WatchDog Timer
#pragma config LVP = OFF //Disabilito Low Voltage Programming
```

Queste sono le 3 righe di codice che troverete sempre nell'incipit di un programma di questo corso! alla direttiva è associato il termine "config" che permette di andare

a modificare alcune configurazioni di base del microcontrollore che rimarranno invariate durante l'esecuzione del programma.

Metodi di programmazione: Il PICkit2

Non ci rimane che capire come possiamo trasferire il programma all'interno del micro.

Questo è il PicKit2:

Il PICkit 2

Le opzioni sono molto varie: possiamo affidarci a programmatori amatoriali che sfruttano la porta parallela e qualche programma installato sul PC per effettuare il download sul microprocessore. Potete altresì sfruttare soluzioni commerciali che propongono varie possibilità. Io vi propongo di utilizzare il PICKIT2. Questo è uno dei programmatori ufficiali di microchip. Tra le varie features che propone di rilevante c'è:

- connessione col PC via USB: utilizzabile anche sui portatili
- Programmazione in-circuit: potete programmare il PIC anche se già montato sulla basetta definitiva, potrete fare modifiche veloci e programmare senza adattatori anche gli SMD
- Debugger: potrete effettuare il debug del programma funzionante direttamente sul micro montato sul circuito
- Logic Analizer: potete analizzare i segnali inviati e ricevuti e comunicazioni seriali

esistono anche dei cloni grazie al fatto che il progetto è stato reso pubblico insieme al firmware, ma il costo dell'originale è basso… tanto vale comprare l'originale, no? :)

Links

Se non l'avete ancora letta andate alla **PRIMA LEZIONE**

Altrimenti passate alla **PROSSIMA LEZIONE**

Per il momento altri articoli li trovate sul **mio sito**:

Dreamsearcher

Presto verranno portate anche quì su ELECTROYOU!

Estratto da "http://www.electroyou.it/mediawiki/index.php?title=UsersPages:Crestus:tutorial-pic-la-programmazione"