Esercitazioni C Enhanced Midrange

12 – Funzioni analogiche: ADC.

Un modulo analogico di sensibile importanza è l'**ADC** – *Analog to Digital Converter*, ovvero convertitore da analogico a digitale. Il suo scopo è trasformare una tensione analogica in un valore binario, calcolabile dal microcontroller.

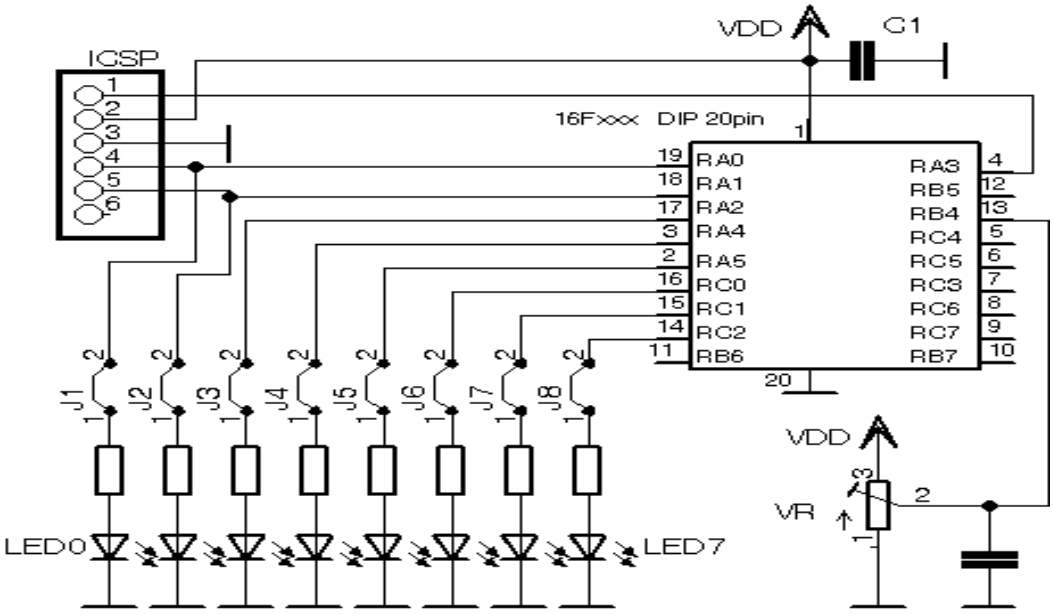
Sia 12F1572 che 16F1619 dispongono di ADC con risoluzione a 10bit ($2^{10} = 1024$).
(peraltro, esistono anche PIC con ADC a 8 e a 12bit).

Vediamo come sperimentare il funzionamento dell' ADC.

Qui maggiori informazioni sul modulo ADC.

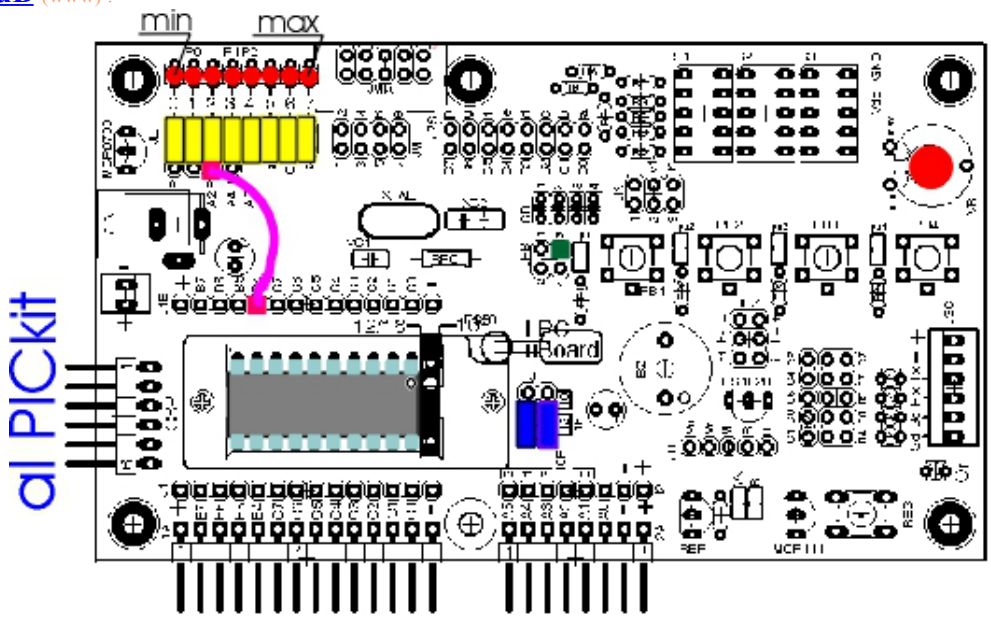
12.1 – Il circuito.

Iniziamo con l'effettuare una conversione della tensione sul cursore del potenziometro disponibile sulla scheda di sviluppo e visualizziamo il risultato su una barra di 8 LED, genere VU meter..



Il cursore del potenziometro è collegato a RB4 che è AN10.
I LED da 0 a 7 costituiscono una barra che incrementa con il variare della tensione sul cursore di VR. Non usiamo il reset MCLR.

Sulla **LPCuB** (www) :



Ricordiamo, per realizzazioni su supporto diverso, che il LED collegati a **RA0** e **RA1** devono essere a bassa corrente oppure essere rimossi durante la programmazione.

Il potenziometro fornisce al cursore una tensione variabile tra **Vdd** e **Vss** e questa viene inviata ad un ingresso analogico del convertitore AD.

Per quanto riguarda la stesura del testo sorgente, per prima cosa osserviamo che i LED sono collegati ad occupare tutte le uscite utili di **PORTA** (**RA01,2,4,5**) e tre pin di **PORTC**, per cui utilizziamo come ingresso analogico **RB4** che è un pin di **PORTB**, ovvero **AN10**
Il ponticello volante "viola" collega il cursore con **AN10-RB4.**

Come abbiamo visto in precedenza, la numerazione dei pin dei port e la numerazione dei canali analogici non sono allineate.
Se inizialmente abbiamo **RA0=AN0**, **RA1=AN1**, già non possiamo associare **AN3** con **RA3** che è un pin speciale, dato che ospita la funzione **MCLR**, per cui **AN3** sarà associato a **RA4** e così via.
Inoltre, in alcuni port solo una parte dei pin può assumere funzioni analogiche.
Ne risulta che, ad esempio nel nostro caso, **RB4** in **ANSELB** è **ANSB4**, ma è il canale **AN10** dell'ADC.

Quindi:

```
ANSELB = 0b00010000 ;          // RB4 analogico AN10
```

e non dobbiamo scordarci di rimuovere il weak pull-up dall'ingresso analogico:

```
WPUBbits.WPUB4 = 0;            // no WPU su RB4
```

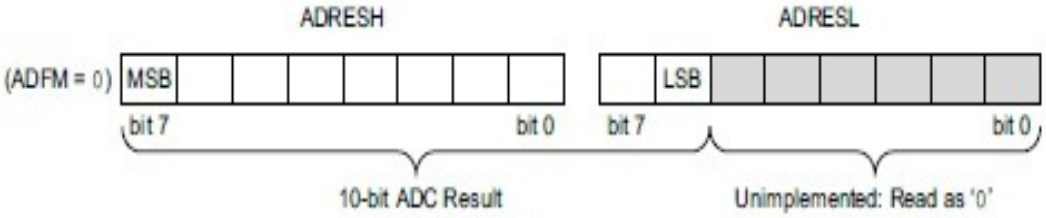
Possiamo configurare l'ADC:

```
ADCON1bits.ADCS = 0b000;       // Fosc/2 ->Tad = 2*Tosc = 4us @ 500kHz
ADCON1bits.ADFM = 0;           // left just.
ADCON1bits.ADPREF = 0b00;      // Vref+ da Vdd
ADCON0bits.CHS = 0b01010;      // seleziona AN10
ADCON0bits.ADON = 1;           // ADC on
```

Dato che il clock principale è minore di 1MHz, possiamo usare tranquillamente un **Fosc/2**, con un tempo di conversione per bit di 4us.

La tensione di riferimento è semplicemente la Vdd. Non ha importanza il suo valore: il PIC può operare da 2.5 a 5.5V e, dato che il risultato della conversione non rappresenta che una frazione del valore della Vdd, non importa quale sia l'alimentazione. L'unico effetto di una bassa Vdd è una riduzione della luminosità dei LED in uscita.
Comunque, intendiamo gli esempi presentati, salvo diversa indicazione, come alimentati a 5V.

Selezioniamo la giustificazione a sinistra per avere gli 8 bit più significativi in **ADRESH**.



In questo modo abbiamo che la lettura del registro **ADRESH** renderà gli 8 bit più significativi della conversione, mentre i due meno significativi saranno contenuti in **ADRESL**.

Il loop principale consiste nella conversione:

```
__delay_us(10);              // tempo acquisizione 10 us
  ADCON0bits.GO = 1;         // avvia conversione
  while (ADCON0bits.GO);     // attesa fine conversione
```

Iniziamo con il tempo di acquisizione, fissato arbitrariamente a 10us. Questo garantisce che il condensatore di campionamento sia caricato opportunamente.

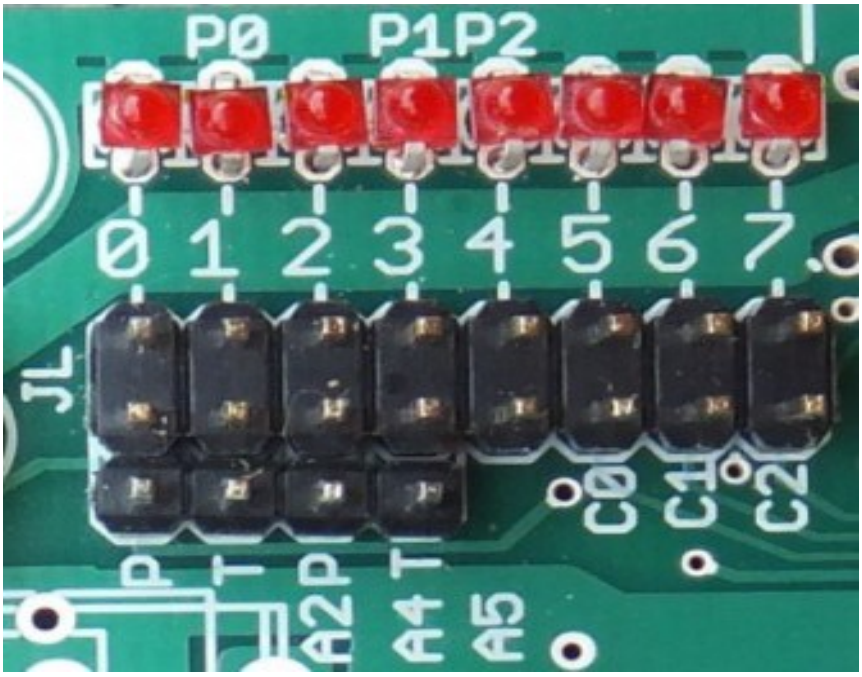
Quindi avviamo la conversione con il settaggio del bit **GO**.

Restiamo poi in attesa che la fine della conversione riporti a 0 in bit.

A questo punto abbiamo gli 8 bit più significativi in **ADRESH** e i due meno significativi in **ADRESL**. Se non consideriamo questi ultimi, avremo, in sostanza, ridotto il risultato ad una conversione a 8 bit. Da oservare che l'eliminazione dei due bit meno significativi in generale stabilizza il risultato della conversione che, in presenza di rumore elettrico o disturbi sul segnale da convertire, potrebbe trovarsi con l'ultimo bit, o i due ultimi bit, di valore instabile.

Si pone, ora, il problema di trasformare questi 8 bit in una barra di LED.

Come visto in un esempio precedente, sulla scheda di sviluppo **LPCuB** (www) abbiamo questa disposizione dei LED/pin:



Abbiamo già usato questi LED come contatore binario. Ora vogliamo accenderli come una barra

che cresce da sinistra a destra in relazione all'aumentare della tensione all'ingresso del convertitore.

Abbiamo la seguente corrispondenza:

LED 0	LED 1	LED 2	LED 3	LED 4	LED 5	LED 6	LED 7
RA0	**RA1**	**RA2**	**RA4**	**RA5**	**RC0**	**RC1**	**RC2**

Vogliamo ottenere una barra del genere di quella dei VUmeters con il minimo indicato da LED0 e il massimo da LED7. La barra parte da LED0 e si allunga verso LED7 all'aumentare del valore del risultato della conversione, ovvero della tensione di ingresso, e viceversa.

La topologia della sequenza dei bit è già favorevole, ma abbiamo due intoppi:

- **RA3** non può essere una uscita, quindi occorre "saltarlo"
- parte dei bit dipendo da **PORTA** e parte da **PORTC**

La cosa è risolvibile, attraverso le istruzioni del programma, con meno contorsioni di quanto si pensi.

Per prima cosa vediamo come assegnare valori ai vari LED accesi.
Abbiamo ridotto la conversione con un risultato a 10 bit (1024 step) in 8 bit (256 step), escludendo i due meno significativi di **ADRESL**. Da un punto di vista matematico abbiamo diviso il risultato per 4.

Possiamo trattare il risultato in un solo byte (che può contenere valori da 0 a 255).
Se rappresentiamo il valore su una barra a 8 LED, possiamo assegnare ad ogni LED un ottavo del valore di fondo scala, quindi 256/8 = 32.

Di conseguenza:

- il primo LED si accende se il risultato è >= 32
- il secondo LED si accende se il risultato è >=64
- il terzo LED si accende se il risultato è >=96
- il quarto LED si accende se il risultato è >=128
- il quinto LED si accende se il risultato è >=160
- il sesto LED si accende se il risultato è >=192
- il settimo LED si accende se il risultato è >=224
- l'ottavo LED si accende se il risultato è =255

In istruzioni, una via semplice corrisponde ad una serie di **if**:

```
unsigned char  barC = 0;      // bit della barra su PORTC
unsigned char  barA = 0;      // bit della barra su PORTA

      if (ADRESH>=32) {
       barA = 1;              // acceso LED0
      }
      if (ADRESH>=64) {
       barA = 3;              // accesi LED0 e 1
      }
      if (ADRESH>=96) {
```

```
    barA = 7;            // accesi LED0, 1, 2
  }
  if (ADRESH>=128) {
   barA = 0x17;          // accesi LED0, 1, 2, 3
  }
  if (ADRESH>=160) {
   barA = 0x37;          // accesi LED0, 1, 2, 3, 4
  }
  if (ADRESH>=192) {
    barC = 0x1;          // acceso LED5
  }
  if (ADRESH>=224) {
   barC = 0x3;           // acceso LED6
  }
  if (ADRESH=255) {
   barC = 0x7;           // acceso LED7
  }
  LATC = barC;
  LATA = barA;
```

Dato che i bit da mandare a 1 per accendere i LED si trovano su due port differenti, utilizziamo due locazioni di RAM per creare due registri in cui scrivere lo stato voluto dei bit. Quando l'immagine della barra è completa nei due registri, li copiamo nei port.

Da notare che copiamo direttamente i registri nei port senza una maschera intermedia e questo azzera tutti gli altri bit non utilizzati per i LED. Per **PORTA**, che è interamente utilizzato, la soluzione è adeguata, ma se avessimo impegnati i bit 7:3 di **PORTC** in altre applicazioni, dovremmo ricorrere ad una scrittura più complessa, in modo da variare solo i bit interessati alla realizzazione della barra senza modificare gli altri, creando una maschera prima di scrivere nel port. Di questo va tenuto conto in altre applicazioni dove i bit di uno stesso port sono assegnati a compiti differenti..

Il risultato del programma sarà il seguente: ruotando il potenziometro, vedremo estendersi o ridursi la barra a seconda del valore di tensione presente sul cursore. E' la base di VU-meter.

Se dobbiamo dare un valore numerico ai LED, dobbiamo considerare che la risoluzione della conversione è ***Vdd/256 = 5 /256=0.0195mV***.
Ogni LED si accende ad un aumento di ***32*0.0195mV=0.625V***. Ad esempio, con 5 led accesi avremo sul cursore del potenziometro una tensione di ***0.625V * 5 = 3.125V***.

La visualizzazione potrebbe anche esse a punto invece che a barra, adeguando i valori selezionati dagli **if**.

es12C

```
/*******************************************************************
*-------------------------------------------------------------------
*
*    Titolo        :  C Enhanced - es12C
*                     Sperimentazione su ADC.
*                     Uscita conversione su 8 LED
*    Data          :  02-10-2019
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  PIC16F1619
*    Autore        :  afg
*-------------------------------------------------------------------
*******************************************************************
*  Impiego pin :
*  ---------------
*       16F1619 @ 20 pin
*
*                  |__\/__|
*             Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*             RC5 -|5   16|- RCO
*             RC4 -|6   15|- RC1
*             RC3 -|7   14|- RC2
*             RC6 -|8   13|- RB4
*             RC7 -|9   12|- RB5
*             RB7 -|10  11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: Out LED4
*    RA4      3: Out LED3
*    RA3      4: In
*    RC5      5: In
*    RC4      6: In
*    RC3      7: In
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: An  AN10
*    RC2     14: Out LED7
*    RC1     15: Out LED6
*    RC0     16: Out LED5
*    RA2     17: Out LED2
*    RA1     18: Out LED1
*    RA0     19: Out LED0
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz
*******************************************************************/
```

```
#include "C:\Corso_C\MyIncludes\conf1619_1.h"
#include <xc.h>

#define _XTAL_FREQ  500000

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

/* Inizializza hardware */

    ANSELA = 0;
    ANSELC = 0;
    ANSELB = 0b00010000 ;           // RB4 analogico AN10

    TRISA = 0;                      // PORTA uscite
    TRISC = 0;                      // PORTC uscite
                                    // PORTB ingressi
    LATA = 0;                       // LED spenti
    LATC = 0;

    WPUBbits.WPUB4 = 0;             // no WPU su RB4
    OPTION_REGbits.nWPUEN = 0;      // abilita WPU per altri input

// configura ADC
    ADCON1bits.ADCS = 0b000;        // Fosc/2 ->Tad = 2*Tosc = 4us @ 500kHz
    ADCON1bits.ADFM = 0;            // left just.
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b01010;       // seleziona AN10
    ADCON0bits.ADON = 1;            // ADC on


    /*** Main loop ***/
    while (1) {

    // campiona ingresso ADC
      __delay_us(10);               // tempo acquisizione 10 us
      ADCON0bits.GO = 1;            // avvia conversione
      while (ADCON0bits.GO);        // attesa fine conversione

    // display del risultato su 8 LED divisi tra PORTA e PORTC
unsigned char  barC = 0;            // bit della barra su PORTC
unsigned char  barA = 0;            // bit della barra su PORTA

      if (ADRESH>=32) {
       barA = 1;
      }
      if (ADRESH>=64) {
       barA = 3;
      }
      if (ADRESH>=96) {
       barA = 7;
      }
      if (ADRESH>=128) {
       barA = 0x17;
      }
      if (ADRESH>=160) {
       barA = 0x37;
      }
      if (ADRESH>=192) {
```

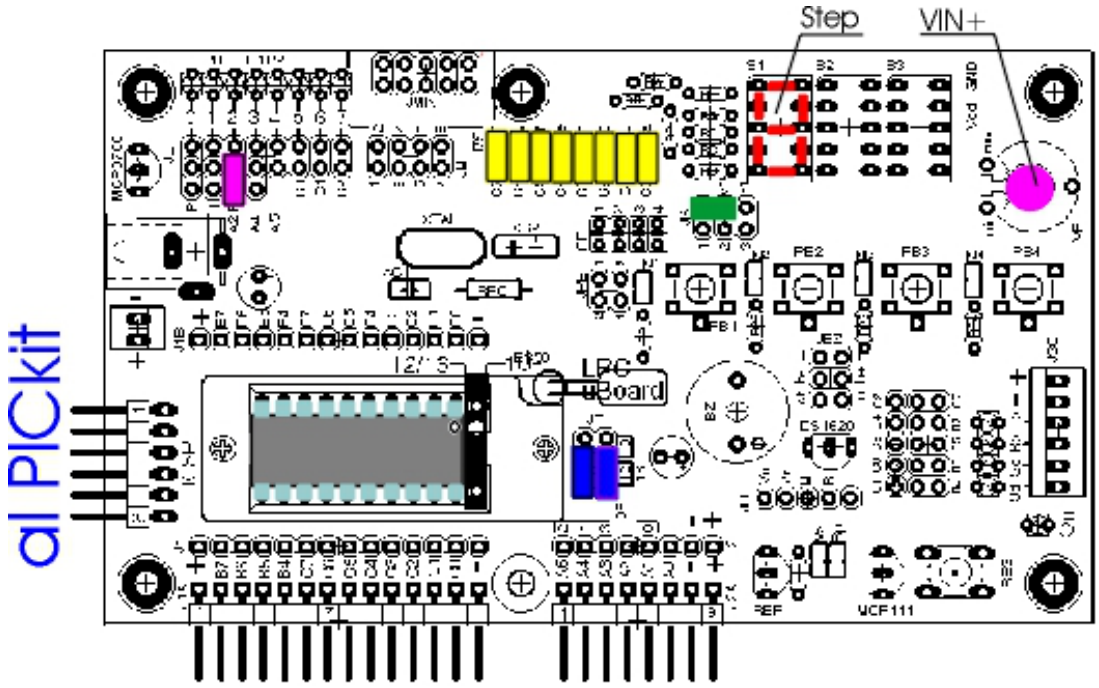
```
        barC = 0x1;
       }
       if (ADRESH>=224) {
        barC = 0x3;
       }
       if (ADRESH>=250) {
        barC = 0x7;
       }
       LATC = barC;
       LATA = barA;

  }
}
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC16F1619).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es12C.***

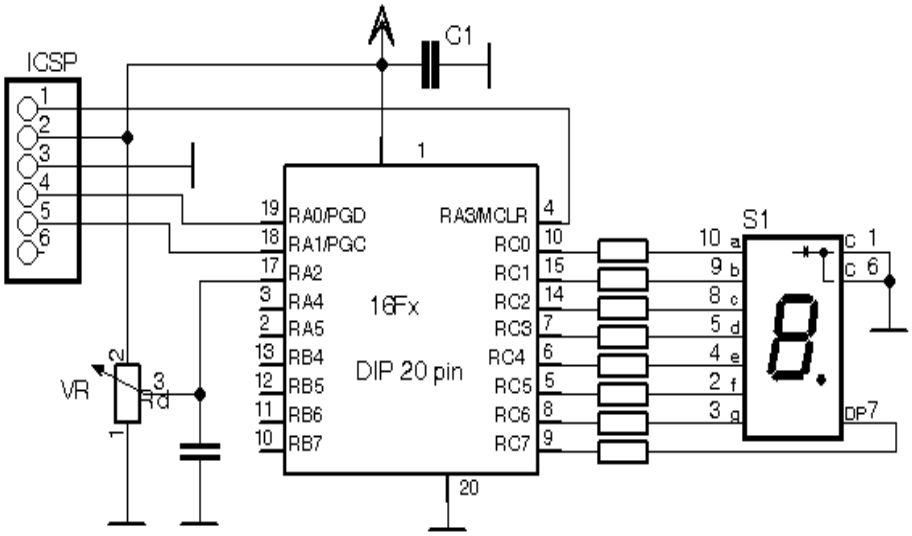
12.2 – Variazione I.

Avendo a disposizione un display a 7 segmenti possiamo utilizzarlo come visualizzatore del risultato.

Sulla scheda di sviluppo abbiamo i segmenti collegati al **PORTC**, per cui posiamo utilizzare **RA2** come ingresso analogico.



Se serve lo schema elettrico:



Il sorgente è del tutto analogo al precedente: cambia solo la sezione che trasferisce il risultato della conversione al display.
Sul display a 7 segmenti possiamo presentare facilmente le 16 cifre esadecimali, da 0 a F, per cui,

come visto in precedenza, ci occorre una tabella di conversione.

Consideriamo, come nell'esempio precedente, di utilizzare solamente 8 bit del risultato della conversione, in **ADRESH** . Se utilizziamo le cifre esadecimali da 0 a F, per ogni cifra presentata sul display avremo 1/16 del risultato. Quindi, dividiamo questo per 16 per ottenere il puntatore della tabella:

```
// display del risultato sul display
   tab7segm = ADRESH/16 ;
```

Le restanti istruzioni non dovrebbero aver bisogno di spiegazione.

Sul display avremo cifre da 0 a F, a seconda del valore risultante dalla conversione.

Se dobbiamo dare un valore numerico alle cifre, dobbiamo considerare che la risoluzione della conversione è ***Vdd/256 = 5 /256=0.0195mV***.
Ogni cifra che si accende corrisponde ad un aumento di ***16 *0.0195mV=0.3125V***.
Ad esempio, con la cifra 5 accesa accesi avremo sul cursore del potenziometro una tensione di ***0.3125V * 5 = 1.56V***.

Nel considerare la conversione AD dobbiamo tenere presenti alcuni punti:

- **La precisione della conversione dipende essenzialmente dalla precisione della tensione di riferimento**.
- La Vdd è la scelta più semplice per la tensione di riferimento, ma non la migliore, dato che la sua precisione non ha molta importanza per l'alimentazione generale del processore, ma è fondamentale rispetto alla conversione. Se la Vdd ha un valore diverso da 5V, le frazioni assegnate ai risultati della conversione vanno ricalcolate di conseguenza.
- Inoltre, solitamente, la Vdd è affetta da disturbi; anche con una alimentazione a batteria è probabile che ci sia rumore elettrico generato dallo stesso microcontroller con le sue commutazioni veloci.

Solitamente, un risultato instabile della conversione è dovuto a problemi sulla Vdd o sul segnale da convertire, oppure si ha una impedenza troppo elevata del segnale o, ancora, un tempo di acquisizione insufficiente.

La soluzione comune è quella di eliminare i bit meno significativi oppure mediare diverse letture consecutive. Questi, però, sono metodi empirici; la soluzione ideale è quella di cercare la causa dell'instabilità e correggerla.

Ad esempio, usando una tensione di riferimento della conversione diversa dalla Vdd, abbassare l'impedenza del segnale da convertire, con un buffer o anche applicando una capacità al pin di ingresso; dove necessario, effettuare la conversione quando altre azioni interne al micro, come ad esempio il multiplex di carichi esterni, con i relativi picchi di corrente, sono sospesi, ecc.

es12C_1

```
/*******************************************************************
*-------------------------------------------------------------------
*
*    Titolo        :  C Enhanced - es12C_1
*                     Sperimentazione su ADC.
*                     Uscita conversione su 1 display 7 segm
*    Data          :  02-10-2019
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  PIC16F1619
*    Autore        :  afg
*-------------------------------------------------------------------
*******************************************************************
*  Impiego pin :
*  ----------------
*       16F1619 @ 20 pin
*
*                  |__\/__|
*             Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*             RC5 -|5   16|- RCO
*             RC4 -|6   15|- RC1
*             RC3 -|7   14|- RC2
*             RC6 -|8   13|- RB4
*             RC7 -|9   12|- RB5
*             RB7 -|10  11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: In
*    RA4      3: In
*    RA3      4: In
*    RC5      5: In  segm f
*    RC4      6: In  segm e
*    RC3      7: In  segm d
*    RC6      8: In  segm g
*    RC7      9: In  segm dp
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In
*    RC2     14: Out segm c
*    RC1     15: Out segm b
*    RC0     16: Out segm a
*    RA2     17: An  AN2
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz
*******************************************************************/

#include "C:\Corso_C\MyIncludes\conf1619_1.h"
```

```
#include <xc.h>

#define _XTAL_FREQ  500000

#define SegmPort  LATC           // PORTC comando segmenti

/***** PROTOTIPI *****/
// display digit on 7 segment
void tab7seg(char digit);

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

/* Inizializza hardware */
    ANSELA = 0b00000100 ;           // solo RA2 analogico
    ANSELB = 0;
    ANSELC = 0;

    TRISC = 0;                      // PORTC uscite
    TRISB = 0;                      // PORTB uscite
    LATC = 0;                       // LED spenti
    LATB = 0;

    WPUAbits.WPUA2 = 0;             // no WPU su RA2
    OPTION_REGbits.nWPUEN = 0;      // abilita WPU per altri input

// configura ADC
    ADCON1bits.ADCS = 0b000;        // Fosc/2 ->Tad = 2*Tosc = 4us @ 500kHz
    ADCON1bits.ADFM = 0;            // left just.
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b00010;       // selziona AN2
    ADCON0bits.ADON = 1;            // ADC on


    /*** Main loop ***/
    while (1) {

    // campiona ingresso ADC
      __delay_us(10);               // tempo acquisizione 10 us
      ADCON0bits.GO = 1;            // avvia conversione
      while (ADCON0bits.GO);        // attesa fine conversione

    // display del risultato sul display
      //digit = (ADRESH/16);
      tab7seg(ADRESH/16) ;
    }                               // chiusura while
}                               // chiusura main

    /**** FUNZIONI ****/
/* 7 segments lookup table per cifre da 0 a 9
   Rx0=segm.a, Rx1= segm.b, ...Rx7= segm.dp */
 void tab7seg (unsigned char digit){
  const unsigned char mask7seg[16] = {
        0x3F,  // 0
        0x06,  // 1
        0x5B,  // 2
        0x4F,  // 3
        0x66,  // 4
```

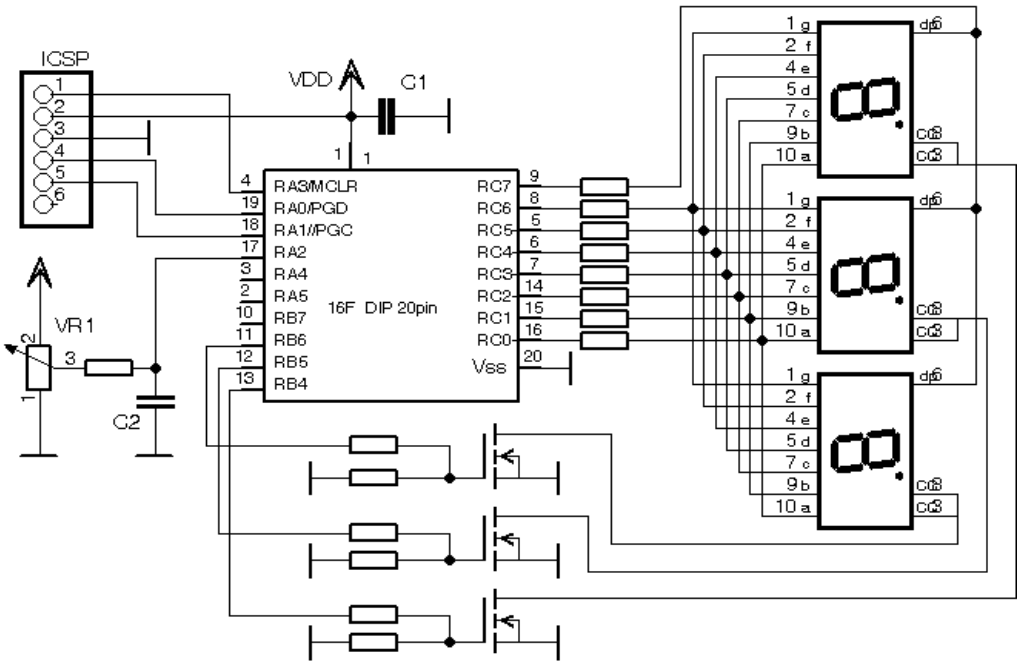
```
        0x6D,  // 5
        0x7D,  // 6
        0x07,  // 7
        0x7F,  // 8
        0x6F,  // 9
        0x77,  // A
        0x7C,  // b
        0x39,  // C
        0x5E,  // d
        0x79,  // E
        0x71   // F
  };                            // chiusura tabella
  SegmPort =  mask7seg[digit];
 }                              // chiusura funzione
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC16F1619).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es12C_1.***

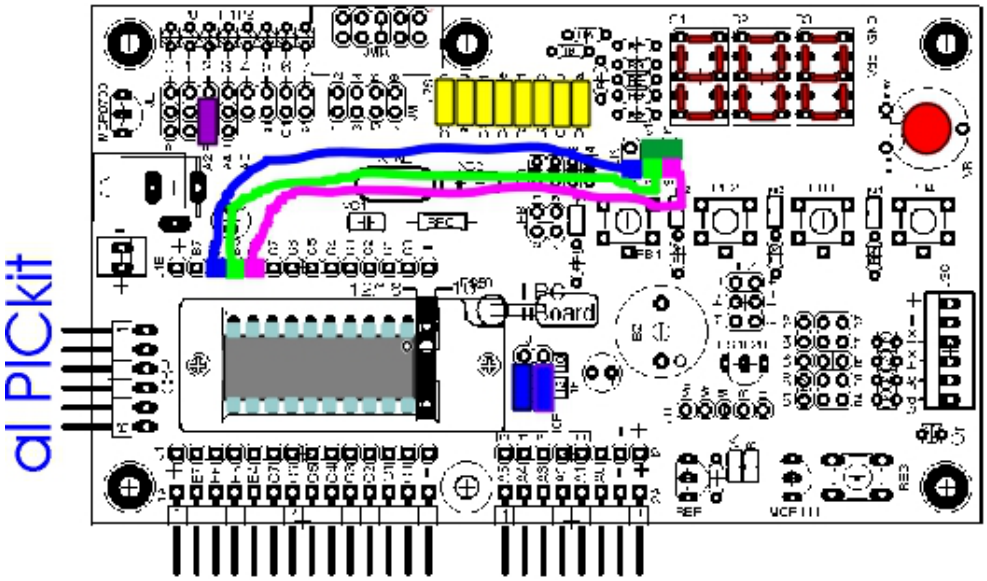
12.3 – Variazione II.

Sulla scheda di sviluppo abbiamo un totale di 3 display: i 10 bit della conversione al massimo corrispondono all'esadecimale 0x3FF (decimale 1023), quindi rappresentabili nel display.

Lo schema elettrico, già visto:



Disposizione sulla scheda :



Combiniamo, quindi, quello che conosciamo sulla gestione del multiplex e sull'uso dell'ADC.

Come già fatto in un esempio precedente, utilizziamo tutto il **PORTC** per comandare i segmenti e tre

bit del **PORTB** per comandare i catodi dei display.
La già vista lookup table sarà necessaria per passare da valore a maschera dei segmenti, utilizzando le cifre esadecimali da 0 ad F

```
// 7 segments lookup table per cifre da 0 a F
// Rx0=segm.a, Rx1= segm.b, ...Rx7= segm.dp
  const unsigned char tab7seg[16] = {
        0x3F,  // 0
        0x06,  // 1
        0x5B,  // 2
        0x4F,  // 3
        0x66,  // 4
        0x6D,  // 5
        0x7D,  // 6
        0x07,  // 7
        0x7F,  // 8
        0x6F,  // 9
        0x77,  // A
        0x7C,  // b
        0x39,  // C
        0x5E,  // d
        0x79,  // E
        0x71   // F
  };
```

Definiamo anche tre variabili per le cifre da inviare al display e una addizionale per indicare che i dati della conversione sono disponibili e che ci servirà per la gestione dell'intterupt.

```
/***** VARIABILI *****/
unsigned char  unit = 0;        // unità
unsigned char  dec = 0;         // decine
unsigned char  cent = 0;        // centinaia
unsigned char  addready=0;      // flag dati adc pronti
```

Le variabili sono inizializzate a 0 per il primo loop della presentazione sui display.

Abbiamo l'ingresso analogico su **RA2**, dal quale disabilitiamo il weak pull-up.

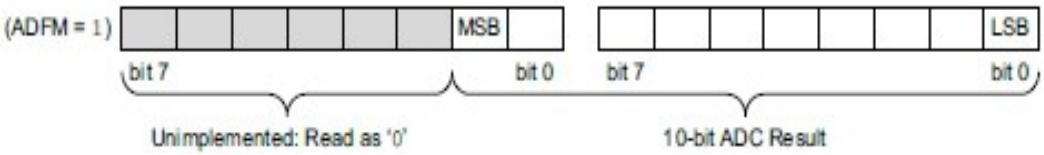
```
     ANSELA = 0b00000100;            // RA2 analogico AN2
     WPUAbits.WPUA2 = 0;             // no WPU su RA2
```

Portiamo il clock a 2MHz e impostiamo il Timer0 per avere un overflow ogni 2ms circa (esattamente 2.048ms). Come già visto in precedenza il timer opera in free run dato che non ci occorre una particolare precisione: l'overflow viene usato solamente per il refresh del multiplex, il quale ha come unica esigenza quella di non diventare troppo lento ed evitare lo sfarfallio delle cifre.

Impostiamo l'ADC per l'ingresso su **AN2** e per la giustificazione a destra del risultato:

```
  ADCON1bits.ADFM = 1;            // right justified
  ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
  ADCON0bits.CHS = 0b00010;       // selziona AN2
```

In questo modo abbiamo i due bit più significativi in **ADRESH** e gli altri 8 in **ADRESL**, cosa che ci facilita la scomposizione esadecimale.



Il **main** si limita ad avviare la conversione: quando una conversione è completata viene settato il flag indicatore:

```
addready = 1;        // dati pronti
```

Il lavoro di passaggio dei dati al display lo svolge la routine di interrupt, con il metodo di switch già visto:

```
Switch (stepcntr){
    case 0:
        KC=0;                   // spegne display precedente
        LATC = tab7seg[unit];   // output LSB
        KU = 1;                 // accende display attuale
        break;
```

Stessa struttura vista in precedenza, con la gestione dei comandi dei catodi per la cifra in corso di presentazione. Il **PORTC** che comanda i display viene caricato con la maschera ottenuta dalla lookup table, con indice pari al valore da presentare.

Questo valore è ottenuto scomponendo il risultato della conversione in mezzi bytes.

```
unit = ADRESL & 0x0F ;
dec  = (ADRESL & 0xF0)>>4 ;
cent = ADRESH;
```

Se non fosse chiaro, occorre ricordare che una cifra esadecimale condensa 4 bit binari; un byte sarà rappresentato da due cifre esadecimali:

nibble alto				**nibble basso**			
7	6	5	4	3	2	1	0
ADRESL							

Con *nibble* si identifica un gruppo di 4 bit, pari ad una metà di un byte. Per cui avremo un upper nibble (nibble alto per i bit da 7 a 4) e un lower nibble (per i bit da 3 a 0).

Ad esempio, se il numero binario di 8 cifre è 0b10100101 il suo corrispondente esadecimale è 0xA5

	nibble alto	**nibble basso**

bit	7	6	5	4	3	2	1	0
binario	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
hex	**A**				**5**			

Quindi, scomponendo il risultato della conversione in nibble, abbiamo i valori esadecimali di ogni cifra da passare alla tabella.

Ricordiamo che in qualsiasi caso i numeri contenuti nel microcontroller sono dati binari costituiti da 1 e 0.
L'impiego di una notazione esadecimale ci facilita la comprensione delle azioni svolte su blochi di 4 bit (intendendo di avere acquisito una sufficiente dimestichezza con le varie rappresentazioni...).

Nel dettaglio ulteriore:

```
unit = ADRESL & 0x0F ;          // solo il nibble basso
```

con l'AND otteniamo un valore pari a **0000nibblebasso**. Ovvero azzeriamo il nibble alto. Quindi se il numero binario del risultato è **0x2AC**, **unit** sarà uguale a **0x0C**.

```
dec  = (ADRESL & 0xF0)>>4 ;    // solo il nibble alto
```

che dobbiamo shiftare di 4 posti per ottenere **0000nibblealto**. Quindi se il numero binario del risultato è **0x2AC**, **dec** sarà uguale a **0x0A**.

```
cent = ADRESH;
```

Nessuna operazione necessaria per i due bit di **ADRESH**, che sono già nella forma adatta per essere passati alla tabella.
Se ci sono ancora dubbia, ricordiamo che la lookup table è prevista per accettare in indice del valore massimo di 16, che sono rappresentati da 4 bit binari, ovvero dal solo lower nibble di un byte.

16 decimale = 0x0F esadecimale = 0b00001111 binario.

Il passaggio del risultato della conversione AD in segmenti viene effettuata solamente se il flag di dati pronti è stato settato dalla fine della conversione AD.

```
if (addready){
```

flag che alla fine dell'operazione viene cancellato. Sarà settato dalla conclusione della prossima conversione.

Inseriamo questo semplice meccanismo nell'ultimo step del multiplex. La ragione (una volta che ci abbiamo pensato) è semplice – genere “elementare, Watson...”.
Dobbiamo considerare che la conversione AD e la presentazione sul display **sono del tutto**

asincrone tra di loro.
Appena il programma è attivato, nessuna conversione è stata effettuata e il dato a disposizione del display sarà 0, come abbiamo inizializzato le variabili relative.
Questa situazione permane fino a che una conversione è effettuata.
Il meccanismo del multiplex, però, prende in carico i nuovi valori solamente dopo aver effettuato la scansione di tutte e tre le cifre, per evitare di presentare risultati parziali; a questo punto i nuovi valori saranno quelli della conversione del ciclo successivo.
Usando questo sistema buffer + flag non può capitare che il risultato di una conversione vada a mescolarsi a quello precedente che non è ancora stato completamente inviato al display.

Il risultato sarà quello di ottenere sul display valori da 00 a 3F a seconda della posizione del cursore del potenziometro.

---

es12C_2

```
/*******************************************************************
*-------------------------------------------------------------------
*
*    Titolo        :  C Enhanced - es12C_2
*                     Sperimentazione su ADC.
*                     Uscita conversione su 3 display 7 segm in hex
*    Data          :  02-10-2019
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  PIC16F1619
*    Autore        :  afg
*-------------------------------------------------------------------
*******************************************************************
*  Impiego pin :
*  ----------------
*       16F1619 @ 20 pin
*
*                   __\/__
*             Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*             RC5 -|5   16|- RCO
*             RC4 -|6   15|- RC1
*             RC3 -|7   14|- RC2
*             RC6 -|8   13|- RB4
*             RC7 -|9   12|- RB5
*             RB7 -|10  11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: In
*    RA4      3: In
*    RA3      4: In
*    RC5      5: In  segm f
*    RC4      6: In  segm e
*    RC3      7: In  segm d
*    RC6      8: In  segm g
*    RC7      9: In  segm dp
*    RB7     10: In
*    RB6     11: Out K2
*    RB5     12: Out K1
*    RB4     13: Out K0
*    RC2     14: Out segm c
*    RC1     15: Out segm b
*    RC0     16: Out segm a
*    RA2     17: An  AN2
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz
*******************************************************************/

#include "C:\Corso_C\MyIncludes\conf1619_1.h"

#include <xc.h>
```

```
#define _XTAL_FREQ  2000000

// Pin assignments
#define KC   LATBbits.LATB6        // katodo centinaia
#define KD   LATBbits.LATB5        // katodo decine
#define KU   LATBbits.LATB4        // katodo unità
#define SegPort LATC               // port comando segmenti

// 7 segments lookup table per cifre da 0 a F
// Rx0=segm.a, Rx1= segm.b, ...Rx7= segm.dp
  const unsigned char tab7seg[16] = {
        0x3F,  // 0
        0x06,  // 1
        0x5B,  // 2
        0x4F,  // 3
        0x66,  // 4
        0x6D,  // 5
        0x7D,  // 6
        0x07,  // 7
        0x7F,  // 8
        0x6F,  // 9
        0x77,  // A
        0x7C,  // b
        0x39,  // C
        0x5E,  // d
        0x79,  // E
        0x71   // F
  };                           // chiusura tabella

/***** VARIABILI *****/
unsigned char  unit = 0;       // unità
unsigned char  dec = 0;        // decine
unsigned char  cent = 0;       // centinaia
unsigned char  addready=0;     // flag dati adc pronti
unsigned char  irqcntr =0;     // contatore interruzioni

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

/* Inizializza hardware */
    OSCCONbits.IRCF = 0b1100 ;      // clock 2MHz

    ANSELA = 0b00000100;            // RA2 analogico AN2
    ANSELC = 0;                     // PORTC no analogca
    ANSELB = 0;                     // PORTB no analogica

    TRISC = 0;                      // PORTC uscite
    TRISB = 0;                      // PORTB uscite
                                    // PORTA ingressi
    LATC = 0;                       // display spenti
    LATB = 0;

    WPUAbits.WPUA2 = 0;             // no WPU su RA2
    OPTION_REGbits.nWPUEN = 0;      // abilita WPU per altri input

// configura Timer0 per 2ms (2.048) con clock 2MHz
    OPTION_REGbits.T0CS = 0;        // clock interno
    OPTION_REGbits.PSA = 0;         // prescaler
```

```
    OPTION_REGbits.PS = 0b001;      // prescale 1:4
                                    //  -> TMR0 overflows every 2.048 ms
// configura ADC
    ADCON1bits.ADCS = 0b011;        // clock FRC
    ADCON1bits.ADFM = 1;            // right justified
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b00010;       // selziona AN2
    ADCON0bits.ADON = 1;            // ADC on

// enable interrupt
    INTCONbits.T0IE = 1;            // abilita Timer0 interrupt
    INTCONbits.GIE = 1;             // abilita interrupt globale

    /*** Main loop ***/
    while (1) {
    // campiona ingresso ADC
    //  GIE = 0;                    // blocca interrupt
       __delay_us(20);              // tempo acquisizione
      ADCON0bits.GO = 1;            // avvia conversione
      while (ADCON0bits.GO);        // attesa fine conversione
    //  GIE = 1;                    // riabilita interrupt
      addready = 1;                 // dati pronti
     }                          // chiusura while
}                               // chiusura main

/***** INTERRUPT SERVICE ROUTINE *****/
// Presente solo interrupt di Timer0

void interrupt irq (void){
    static unsigned char  stepcntr = 0;   // contatore passi

     INTCONbits.T0IF = 0;                  // clear interrupt flag

    // Invia al display il dato seguendo il contatore di passi
    switch (stepcntr){
        case 0:
            KC = 0;                     // spegne display precedente
            SegPort = tab7seg[unit];    // output LSB
            KU = 1;                     // accende display attuale
            break;
        case 1:
            KU = 0;
            SegPort = tab7seg[dec];     // output cifra intermedia;
            KD = 1;
            break;
        case 2:
            KD = 0;
            SegPort = tab7seg[cent];    // output MSB
            KC = 1;
        if (addready){
     // converti risultato ADC in segmenti
         unit = (ADRESL & 0x0F) ;
         dec  = (ADRESL & 0xF0)>>4 ;
         cent = ADRESH;
         addready = 0;              // cancella flag
         }                      // chiusura if
            break;
    }                           // chiusura switch
    // aggiorna contatore per 3 passi e se al massimo, azzera
```

```
    stepcntr++;
    if (stepcntr == 3){
        stepcntr = 0;
    }                           // chiusura if
 }
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC16F1619).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es12C_2.***

12.4 – Una mediazione.

Nell'esempio precedente si rileverà, però un problema: **l'ultima cifra del risultato non sarà stabile**, ma oscillerà attorno ad un valore.

Questo fenomeno è normale in una conversione effettuata nel modo descritto ed è prodotto da diversi fattori:

1. l'errore tipico del convertitore (+/- 1 lsb)
2. il contatto del cursore del potenziometro sulla pista conduttiva
3. l'uso della Vdd come tensione di riferimento
4. la circolazione nel micro e nel circuito stampato di segnali commutati a frequenze elevate

Nel n ostro caso, sono i punti 3 e 4 a determinare i disturbi che influiscono sulla conversione: sostanzialmente, il modulo ADC converte la tensione in ingresso più i disturbi.

E' possibile eliminare, o almeno minimizzare, il problema in diversi modi.
Internamente:

- avendo una Vdd di elevata qualità
- sospendendo le commutazioni più gravose durante la conversione. Ideale è effettuare la conversione AD in modalità sleep, quando ogni altro clock o azione sono congelati

Esternamente si può procedere con la drastica eliminazione dell'ultimo bit del risultato oppure effettuando una media di diverse letture successive: una tecnica può essere quella di sommare quattro letture del risultato della conversione e sciftare a destra 2 volte (ovvero dividere per 4).

In ogni caso, conversioni AD che richiedono una elevata precisione **richiedono anche una elevata cura del software e ancor più dell'hardware**: la qualità della tensione di alimentazione e di quella di riferimento ed anche il layout delle piste digitali e analogiche diventa tanto più importante quanto maggiore è la risoluzione che si vuole ottenere.

Qui, come semplice esempio per verificare l'effetto del rumore interno, possiamo de-commentare le linee che bloccano l'interrupt durante la conversione e ricompilare; si osserverà una certa riduzione dell'instabilità dell'ultima cifra dovuta alla sospensione dei segnali del multiplex.
Data la brevità della conversione AD, la pausa nel ciclo di multiplex non si nota.
Per la prova, nel sorgente dell'esempio precedente, de commentare le linee relative allo spegnimento e alla riaccensione dello switch generale dell'interrupt prima e dopo la conversione e ricompilare.

Possiamo anche provare ad implementare **una semplice media**. Si tratta di aggiungere poche istruzioni.
Essenzialmente, effettuiamo un certo numero di letture, sommiamo i risultati e dividiamo il totale per il numero di letture.

Questo richiede di disporre di alcune registri di appoggio, che andiamo a dichiarare nelle variabili:

```
unsigned int   adval = 0;      // valore media ad
```

```
unsigned int   adres16 = 0;     // analogo ad ADRES
unsigned char  sample = 0;      // numero campionamenti
```

Il registro **adres16** contiene , in un tipo int, i valori **ADRESH:ADRESL**.
Sebbene gli headers di XC8 includano una definizione per un **ADRES** a 16 bit come volatile unsigned short., è consigliabile non utilizzarlo in quanto è possibile che in alcuni headers dei processori la label **ADRES** non sia definita.
La forma seguente è adatta per tutte le situazioni:

```
adres16 = (256U * ADRESH | ADRESL);
```

256U * equivale ad uno shift a sinistra di 8 posizioni di **ADRESH** nel byte alto di **adres16** a cui si aggiunge **ADRESL** come byte basso con un OR. L'indicazione **U** (non obbligatorio, è solo un una buona pratica) conferma al compilatore che si tratta di un numero unsigned.
La cosa può anche essere scritta come:

```
adres16 = (ADRESH<<8 | ADRESL);
```

Il risultato dovrebbe essere identico, dato che il compilatore dovrebbe essere abbastanza furbo per trasformare la moltiplicazione per 256 in 8 shift a destra.
Si potrebbe anche scrivere

```
adres16 = ADRESL;
adres16 |= (ADRESH*256U);
```

ma questo è probabile che generi un numero maggiore di linee Assembly.

Un ulteriore registro serve a totalizzare la somma delle quattro letture, gestite da un ciclo for:

```
while (addready);                  // esegui solo se flag=0
   adval = 0;                     // azzera somma campionamenti
// 4 campionamenti ADC
   for(sample=0; sample<4; sample++){
      __delay_us(10);             // tempo acquisizione
    ADCON0bits.GO = 1;            // avvia conversione
    while (ADCON0bits.GO);        // attesa fine conversione
    adres16 = (256U *ADRESH |ADRESL);  // risultato a 16bit
    adval = (adval + adres16);    // somma campionamenti
   }                              // chiusura for
    adval = adval>>2;             // dividi per 4 la somma
    addready = 1;                 // dati pronti
```

Il loop **for** è eseguito solo se il flag **addready** è azzerato, cosa che abbiamo al primo ingresso.
Si tratta di una variazione sul sistema di switch visto prima per evitare che i dati verso il display siano aggiornati durante un ciclo del multiplex. Solo a ciclo completato è possibili l'aggiornamento dei dati.

Le indicazioni delle parentesi graffe di chiusura sono ancora commentate, per permettere la massima comprensione delle concatenazioni logiche.

Ad ogni passo del loop, il valore ottenuto dalla conversione è aggiunto al precedente

```
adval = (adval + adres16);    // somma campionamenti
```

Dopo 4 campionamenti, il valore della somma viene diviso per quattro con un doppio shift a destra.

```
adval = adval>>2;             // dividi per 4 la somma
```

La ragione della scelta di un numero di campionamenti multiplo di 2 deriva proprio dalla facilità di dividere il risultato per multipli di 2 con un semplice shift.
Potremmo, quindi, anche usare 8 campionamenti con uno shit di tre posizioni, 16 campionamenti con uno shift di 4 posizioni, ecc.

In ogni caso, se si tratta solo di stabilizzare l'ultima cifra e non è voluta per altre ragioni, a**d una media di molte conversioni è sempre preferibile l'eliminazione delle cause hardware che creano l'instabilità.**

In particolare, al posto dello shift, XC8 ci consente di utilizzare anche la forma:

```
adval /= 4;             // dividi per 4 la somma
```

ma è molto probabile che il compilatore generi un numero di istruzioni Assembly molto maggiore di quante necessarie allo shift.

Ora i dati sono pronti per la visualizzazione e questo viene comunicato portando a 1 il flag

```
addready = 1;                // dati pronti
```

Questo consente al terzo passo della routine di interruzione di catturare i dati del prossimo ciclo di multiplex, mentre inibisce un nuovo burst di conversioni.
Le conversioni riprenderanno quando la cattura dei dati è eseguita e il flag viene azzerato.

La routine di interruzione è identica a quanto visto prima, ma l'aggiornamento del valore da presentare sul display è fatto su **`adval`**.

Il risultato è una buona stabilità dell'ultima cifra (a meno di una situazione di cablaggio, segnali e alimentazione disastrosa).

---

es12C_3

```
/*******************************************************************
*-------------------------------------------------------------------
*
*    Titolo        :  C Enhanced - es12C_3
*                     Sperimentazione su ADC.
*                     Uscita conversione su 3 display 7 segm in hex
*                     con media delle letture.
*    Data          :  02-10-2019
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  PIC16F1619
*    Autore        :  afg
*-------------------------------------------------------------------
*******************************************************************
*  Impiego pin :
*  ---------------
*       16F1619 @ 20 pin
*
*                  |__\/__|
*             Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*             RC5 -|5   16|- RCO
*             RC4 -|6   15|- RC1
*             RC3 -|7   14|- RC2
*             RC6 -|8   13|- RB4
*             RC7 -|9   12|- RB5
*             RB7 -|10  11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: In
*    RA4      3: In
*    RA3      4: In
*    RC5      5: In  segm f
*    RC4      6: In  segm e
*    RC3      7: In  segm d
*    RC6      8: In  segm g
*    RC7      9: In  segm dp
*    RB7     10: In
*    RB6     11: Out K2
*    RB5     12: Out K1
*    RB4     13: Out K0
*    RC2     14: Out segm c
*    RC1     15: Out segm b
*    RC0     16: Out segm a
*    RA2     17: An  AN2
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz
*******************************************************************/

#include "C:\Corso_C\MyIncludes\conf1619_0.h"

#include <xc.h>
```

```
#define _XTAL_FREQ  2000000

// Pin assignments
#define KC   LATBbits.LATB6        // katodo centinaia
#define KD   LATBbits.LATB5        // katodo decine
#define KU   LATBbits.LATB4        // katodo unità
#define SegPort LATC               // port comando segmenti

// 7 segments lookup table per cifre da 0 a F
// Rx0=segm.a, Rx1= segm.b, ...Rx7= segm.dp
  const unsigned char tab7seg[16] = {
        0x3F,  // 0
        0x06,  // 1
        0x5B,  // 2
        0x4F,  // 3
        0x66,  // 4
        0x6D,  // 5
        0x7D,  // 6
        0x07,  // 7
        0x7F,  // 8
        0x6F,  // 9
        0x77,  // A
        0x7C,  // b
        0x39,  // C
        0x5E,  // d
        0x79,  // E
        0x71   // F
  };                           // chiusura tabella

/***** VARIABILI *****/
unsigned char  unit = 0;       // unità
unsigned char  dec = 0;        // decine
unsigned char  cent = 0;       // centinaia
unsigned char  addready=0;     // flag dati adc pronti
unsigned char  irqcntr =0;     // contatore interruzioni
unsigned int   adval = 0;      // valore media ad
unsigned int   adres16 = 0;
unsigned char  sample = 0;

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

/* Inizializza hardware */
    OSCCONbits.IRCF = 0b1100 ;      // clock 2MHz

    ANSELA = 0b00000100;            // RA2 analogico AN2
    ANSELC = 0;                     // PORTC no analogca
    ANSELB = 0;                     // PORTB no analogica

    TRISC = 0;                      // PORTC uscite
    TRISB = 0;                      // PORTB uscite
                                    // PORTA ingressi default
    LATC = 0;                       // display spenti
    LATB = 0;

    WPUAbits.WPUA2 = 0;             // no WPU su RA2
    OPTION_REGbits.nWPUEN = 0;      // abilita WPU per altri input
```

```
// configura Timer0 per 2ms (2.048) con clock 2MHz
    OPTION_REGbits.T0CS = 0;        // clock interno
    OPTION_REGbits.PSA = 0;         // prescaler
    OPTION_REGbits.PS = 0b001;      // prescale 1:4
                                    //  -> TMR0 overflows every 2.048 ms
// configura ADC
    ADCON1bits.ADCS = 0b011;        // clock FRC
    ADCON1bits.ADFM = 1;            // right justified
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b00010;       // selziona AN2
    ADCON0bits.ADON = 1;            // ADC on

// enable interrupt
    INTCONbits.T0IE = 1;            // abilita Timer0 interrupt
    INTCONbits.GIE = 1;             // abilita interrupt globale

    /*** Main loop ***/
    while (1) {
    while (addready);               // esegui solo se flag=0
      adval = 0;                    // azzera somma campionamenti
    // 4 campionamenti ADC
      for(sample=0; sample<4; sample++){
          __delay_us(10);           // tempo acquisizione
      ADCON0bits.GO = 1;            // avvia conversione
      while (ADCON0bits.GO);        // attesa fine conversione
      adres16 = (256U *ADRESH |ADRESL);  // risultato a 16bit
      adval = (adval + adres16);    // somma campionamenti
      }                         // chiusura for
      adval = adval>>2;             // dividi per 4 la somma
      addready = 1;                 // dati pronti
     }                          // chiusura while
}                               // chiusura main

/***** INTERRUPT SERVICE ROUTINE *****/
// Presente solo interrupt di Timer0

void interrupt irq (void){

    static unsigned char  stepcntr = 0;   // contatore passi

     INTCONbits.T0IF = 0;                  // clear interrupt flag

    // Invia al display il dato seguendo il contatore di passi
    switch (stepcntr){
        case 0:
            KC = 0;                     // spegne display precedente
            SegPort = tab7seg[unit];    // output LSB
            KU = 1;                     // accende display attuale
            break;
        case 1:
            KU = 0;
            SegPort = tab7seg[dec];     // output cifra intermedia;
            KD = 1;
            break;
        case 2:
            KD = 0;
            SegPort = tab7seg[cent];    // output MSB
            KC = 1;
```

```
        if (addready){                    // solo se dati pronti
    // estrai nibble dal risultato per la tabella hex->segmenti
         unit = (adval & 0x0F) ;          // nibble meno significativo
         dec  = (adval & 0xF0)>>4 ;       // nibble intermedio
         cent = adval>>8;                 // nbble più significativo

    // cancella flag per avviare prossima conversione
         addready = 0;
         }                        // chiusura if
            break;                         //esce dal case2
    }                             // chiusura switch
    // aggiorna contatore per 3 passi del multiplex e se al massimo, azzera
    stepcntr++;
    if (stepcntr == 3){
        stepcntr = 0;
    }                             // chiusura if
 }                                // chiusura interrupt
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC16F1619).
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es12C_3.***

12.5 – Misura della tensione.

Una indicazione in numeri binari non è molto pratica nel caso in cui ci si debba rivolgere ad utenti non strettamente tecnici; usualmente noi utilizziamo numeri decimali.
Quindi, vediamo di convertire il display esadecimale in uno decimale e, visto che stiamo misurando una tensione, provvediamo ad trasformare l'indicazione in volt.

Utilizziamo sempre la configurazione hardware degli esempi precedenti.

La conversione è effettuata a 10 bit, ma, come già fatto in precedenza, utilizziamo solamente gli 8 bit più significativi.
Ne deriva che, utilizzando la tensione di alimentazione come riferimento e supponendo che essa sia 5V, avremo una definizione di 5/255=0.0196V per step di conversione.
Questo ci permette di visualizzare a fondo scala, su tre cifre, 0.0196*255=4.99V.
Notiamo che i valori resi dalla conversione sono forzatamente approssimati, dato che non utilizziamo matematica in virgola mobile e la quantizzazione a 8 bit non consente migliore risoluzione.

In ogni caso, per non scostarci troppo dal valore teorico reso dalla conversione a causa di una perdita di decimali, effettuiamo la media delle conversioni sui 10bit completi, che poi riportiamo a 8 bit

```
adval = adval>>4;                // riduci a 8 bit
```

e convertiamo in volt moltiplicando il risultato della conversione per la risoluzione:

```
adval = (adval*196)/100;         // converti in volt
```

Se **`adval= 0xB5`**, questo indica che il convertitore ha reso 181 passi.
Dato che ognuno vale 19.6mV circa, essi rappresentano la tensione di 181*0.0196=3.5476V.
Abbiamo solo tre cifre disponibili, quindi eliminiamo le ultime 2 dividendo per 100 ed otteniamo 354 (0x162). Il valore ottenuto rappresenta, in numeri binari, la tensione misurata.

Non vogliamo per ora usare numeri a virgola mobile, quindi facciamo corrispondere le centinaia con il primo display, le decine con il secondo e le unità con il terzo e inseriamo il punto decimale tra il primo e il secondo display. Senza ricorre a matematica in floating point abbiamo la visualizzazione corretta.

L'operazione di passare le cifre al display, su 8 bit, è abbastanza semplice, dato che abbiamo a che fare con un numero che non supera le tre cifre:

```
// converti risultato ADC 8 bit in segmenti per 3 display
    unit = (adval %10);
    dec  = (adval/10)%10;
    cent = (adval/100)%10;
```

Il byte viene diviso per 100 per ottenere le centinaia. Il resto è diviso per 10 per ottenere le decine e quanto rimane costituisce le unità.
Ad esempio, supponiamo che il byte contenga 0x162 che è 354 decimale.

`0x162/0x64 = 3` con resto 54 → **centinaia = 3**

`0x054/0x0A = 5` con resto 4 → **decine = 5** → **unità = 4**

Acquisiti i dati da inviare al display, il flag **`addready`** viene azzerato e questo permette una nuova raccolta di dati dell'ADC.

La lettura del display varierà da 000 a 49 .

Notiamo che la le letture hanno una buona stabilità, in quanto sono mediate e utilizzano solamente 8 dei 10bit della risoluzione. Eventuali sfarfallamenti sono principalmente dovuti ad un cattivo contatto del cursore del potenziometro.

---

es12C_4

```
/*******************************************************************
*-------------------------------------------------------------------
*
*    Titolo        :  C Enhanced - es12C_4
*                     Sperimentazione su ADC.
*                     Uscita su 3 display 7 segm con conversione in
*                     tensione
*    Data          :  02-10-2019
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  PIC16F1619
*    Autore        :  afg
*-------------------------------------------------------------------
*******************************************************************
*  Impiego pin :
*  ---------------
*       16F1619 @ 20 pin
*
*                  |__\/__|
*             Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*             RC5 -|5   16|- RCO
*             RC4 -|6   15|- RC1
*             RC3 -|7   14|- RC2
*             RC6 -|8   13|- RB4
*             RC7 -|9   12|- RB5
*             RB7 -|10  11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: In
*    RA4      3: In
*    RA3      4: In
*    RC5      5: In  segm f
*    RC4      6: In  segm e
*    RC3      7: In  segm d
*    RC6      8: In  segm g
*    RC7      9: In  segm dp
*    RB7     10: In
*    RB6     11: Out K2
*    RB5     12: Out K1
*    RB4     13: Out K0
*    RC2     14: Out segm c
*    RC1     15: Out segm b
*    RC0     16: Out segm a
*    RA2     17: An  AN2
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz
*******************************************************************/

#include "C:\Corso_C\MyIncludes\conf1619_0.h"

#include <xc.h>
```

```
#define _XTAL_FREQ  2000000

// Pin assignments
#define KC   LATBbits.LATB6        // katodo centinaia
#define KD   LATBbits.LATB5        // katodo decine
#define KU   LATBbits.LATB4        // katodo unità
#define SegPort LATC               // port comando segmenti
#define dp   0x80                  // dcimal point su RC7

// 7 segments lookup table per cifre da 0 a F
// Rx0=segm.a, Rx1= segm.b, ...Rx7= segm.dp
  const unsigned char tab7seg[10] = {
        0x3F,  // 0
        0x06,  // 1
        0x5B,  // 2
        0x4F,  // 3
        0x66,  // 4
        0x6D,  // 5
        0x7D,  // 6
        0x07,  // 7
        0x7F,  // 8
        0x6F   // 9
        /*0x77,  // A
        0x7C,  // b
        0x39,  // C
        0x5E,  // d
        0x79,  // E
        0x71   // F*/
  };                           // chiusura tabella

/***** VARIABILI *****/
unsigned char  unit = 0;       // unità
unsigned char  dec = 0;        // decine
unsigned char  cent = 0;       // centinaia
unsigned char  addready=0;     // flag dati adc pronti
unsigned char  irqcntr =0;     // contatore interruzioni
unsigned int   adval = 0;      // valore media ad
unsigned int   adres16 = 0;    // 16 bit risultato ADC
unsigned char  sample = 0;     // numero campionamenti

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

/* Inizializza hardware */
    OSCCONbits.IRCF = 0b1100 ;      // clock 2MHz

    ANSELA = 0b00000100;            // RA2 analogico AN2
    ANSELC = 0;                     // PORTC no analogca
    ANSELB = 0;                     // PORTB no analogica

    TRISC = 0;                      // PORTC uscite
    TRISB = 0;                      // PORTB uscite
                                    // PORTA ingressi
    LATC = 0;                       // display spenti
    LATB = 0;

    WPUAbits.WPUA2 = 0;             // no WPU su RA2
    OPTION_REGbits.nWPUEN = 0;      // abilita WPU per altri input
```

```
// configura Timer0 per 2ms (2.048) con clock 2MHz
    OPTION_REGbits.T0CS = 0;        // clock interno
    OPTION_REGbits.PSA = 0;         // prescaler
    OPTION_REGbits.PS = 0b001;      // prescale 1:4
                                    //  -> TMR0 overflows every 2.048 ms
// configura ADC
    ADCON1bits.ADCS = 0b011;        // clock FRC
    ADCON1bits.ADFM = 1;            // right justified
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b00010;       // selziona AN2
    ADCON0bits.ADON = 1;            // ADC on

// enable interrupt
    INTCONbits.T0IE = 1;            // abilita Timer0 interrupt
    INTCONbits.GIE = 1;             // abilita interrupt globale

    /*** Main loop ***/
    while (1) {
    while (addready);
      adval = 0;
    // campiona ingresso ADC
      for(sample=0; sample<4;sample++){
        //GIE = 0;                      // blocca interrupt
        __delay_us(10);                 // tempo acquisizione
      ADCON0bits.GO = 1;                // avvia conversione
      while (ADCON0bits.GO);            // attesa fine conversione
      //GIE = 1;                        // riabilita interrupt
      adres16 = (256U *ADRESH |ADRESL); // unisci in 16 bit
        adval = (adval + adres16);      // somma sample
      }                         // chiusura for
        adval = adval>>4;               // riduci a 8 bit
        adval = (adval*196)/100;        // converti in volt
      addready = 1;                     // dati pronti
     }                          // chiusura while
}                               // chiusura main

/***** INTERRUPT SERVICE ROUTINE *****/
// Presente solo interrupt di Timer0

void interrupt irq (void){

    static unsigned char  stepcntr = 0;    // contatore passi

     INTCONbits.T0IF = 0;                  // clear interrupt flag

    // Invia al display il dato seguendo il contatore di passi
    switch (stepcntr){
        case 0:
            KC = 0;                       // spegne display precedente
            SegPort = tab7seg[unit];      // output LSB
            KU = 1;                       // accende display attuale
            break;
        case 1:
            KU = 0;
            SegPort = tab7seg[dec];       // output cifra intermedia;
            KD = 1;
            break;
        case 2:
            KD = 0;
```

```
            SegPort = tab7seg[cent]|dp;      // output MSB + dp
            KC = 1;
        if (addready){                       // se dati pronti
     // converti risultato ADC 8 bit in segmenti per 3 display
         unit = (adval %10);
         dec  = (adval/10)%10;
         cent = (adval/100)%10;
         addready = 0;                      // cancella flag
         }                       // chiusura if
            break;
    }                            // chiusura switch
    // aggiorna contatore per 3 passi e se al massimo, azzera
    stepcntr++;
    if (stepcntr == 3){
        stepcntr = 0;
    }                            // chiusura if
 }
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC16F1619.
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es12C_4.***

12.5 - PICcolo, ma bello...

In questi esempi abbiamo trascurato il piccolo 12F1572, ma anch'esso dispone di un ADC a 10 bit.

RA0	7	AN0
RA1	6	AN1
RA2	5	AN2
RA3	4	—
RA4	3	AN3
RA5	2	—
VDD	1	—
Vss	8	—

Dato il basso numero di pin, i canali analogici di ingresso al convertitore sono solo quattro, con queste corrispondenze (TABLE 1 – pag 3 del foglio dati).

Osserviamo che, come solito, RA3 non dispone mai di funzione analogica, causa la sua struttura interna di ingresso MCLR.

Anche RA5 non dispone di analogica.

La gestione della funzione di ingresso analogica, per default applicata ai pin, come abbiamo visto, dipende dal registro **ANSELA.** Riepiloghiamo attraverso una immagine la situazione delle label relative.

bit	7	6	5	4	3	2	1	0
I/Odigitale	-	-	**RA5**	**RA4**	**RA3**	**RA2**	**RA2**	**RA0**
ANSELA	-	-	**-**	**ANSA4**	**-**	**ANSA21**	**ANSA1**	**ANSA0**
ADC	-	-	**-**	**AN3**	**-**	**AN2**	**AN1**	**AN0**

Con queste risorse posiamo implementare un indicatore di minima per la tensione di alimentazione del microcontroller.

Supponiamo di avere un sistema alimentato a batteria, con pile alcaline che hanno complessivamente una tensione che varia tra 4.5V a pile cariche e 3.3V a pile scariche. Vogliamo individuare quando le batterie sono da sostituire.

Prima, alcune considerazioni:

- abbiamo all'interno del chip il modulo FVR che fornisce una tensione di riferimento: questa tensione è fortemente indipendente dal valore dall'alimentazione del chip (basta che Vdd sia sufficientemente maggiore di quella generata dal riferimento)
- l'uscita del modulo FVR può essere un ingresso del modulo ADC.

Se abbiamo una tensione di alimentazione che non scende sotto i 2.5V, possiamo ricavare da FVR una tensione di riferimento di 2.048V o anche 1.024V. Questa tensione sarà stabile per tutta la gamma della variazione della Vdd da 2.5 a 5.5V.

Per utilizzare il modulo ADC per valutare la tensione Vdd, si procede in questo modo:

1. programmare il modulo ADC per Vdd/Vss come tensioni di riferimento per la conversione
2. disporre il modulo ADC per la lettura interna di FVR
3. attivare il modulo FVR
4. effettuare la conversione AD

il risultato della conversione indicherà la tensione Vdd.

A prima vista ci si può chiedere come questo sia possibile dato che stiamo misurando la tensione di riferimento interna che è, si deve supporre, assai stabile, costante ed indipendente dalla Vdd.
Ma è proprio su questo punto che si basa la misura, perchè

la **Vref è costante, ma la Vdd è variabile**.

Normalmente la tensione da convertire è variabile e quella del riferimento è fissa. Qui operiamo all'inverso: la tensione da misurare è stabile e quella del riferimento è variabile; vediamo cosa comporta questa situazione.

Quando effettuiamo una conversione AD il risultato è pari a:

risultato conversione = valore da convertire / (Vref / numero degli step)

Ad esempio, se abbiamo disponibile una conversione a 10bit, sono possibili 1024 step. Con una tensione di riferimento di 5V, questo vuol dire che ogni step vale 5/1024 = 4,882 mV per step. Ovviamente, per una misura molto precisa, si cercherà di avere disponibile una tensione di riferimento ben stabile, magari multiplo del numero degli step. Ad esempio, con una Vref da 4096mv ogni step vale 4096/1024 = 4 mV. In questo caso, ad esempio, se la conversione rende 0x64 (100 decimale), il valore letto è di 100 * 4 = 400mV.

Ma osserviamo la cosa da un diverso punto di vista, ovvero quello imposto: valore da convertire stabile e la tensione di riferimento variabile.
Usiamo come Vref+ la Vdd a 5V e come Vref- la Vss a 0V. Se misuriamo la tensione dell' FVR settato a 1.024V abbiamo:

Vref/step = 5000/1024 = 4.882 mV/step

e la conversione rende:

risultato = valore / mV/step = 1024 /4.882= 499,91 decimale -> 0x1F3.

Poniamo ora che la Vdd, usata con Vref+, scenda a 4V; di conseguenza la risoluzione sarà:

Vref / step = 4000/1024 = 3,906 mV/step.

Quindi la lettura del riferimento interno, **che resta costante** a 1024mV renderà:

risultato = valore / mV/step = 1024 / 3,096 = 330,74 -> 0x14A

Se la tensione Vdd scende a 3V:

Vref / step = 3000/1024 = 2,929 mV/step.

Quindi la lettura di 1024mV renderà:

risultato = valore / mV/step = 1024 / 2,929 = 349,60-> 15Dh

In generale:

$$V_{dd} = (\text{step ADC} / \text{risultato conversione}) * V_{fvr}$$

e, nel caso del convertitore AD a 10 bit e la tensione di riferimento a 1.024V:

$$V_{dd} = (1024 / \text{risultato conversione}) * 1.024 = 1/ \text{risultato conversione}$$

Per l' esattezza dovremmo calcolare:

$$V_{dd} = (FVR * ((2^n - 1) / \text{risultato conversione}))$$

ma all' atto pratico possiamo dire che,

con il variare della tensione usata come riferimento per la misura, il risultato della conversione di un valore costante, minore della tensione di riferimento, dà un valore inversamente proporzionale alla tensione di riferimento stessa.

Quindi, usando la Vdd come tensione di riferimento variabile per misurare la FVR costante, dal risultato della conversione possiamo valutare il valore della Vdd.

In questo modo, con una semplice conversione ADC e senza utilizzare alcun pin esterno addizionale, è possibile risalire al valore della Vdd e determinare azioni dipendenti da essa.

Possiamo stendere una tabella per l' area di valori che ci interessano. Ad esempio, con un FVR di 1.024V abbiamo i seguenti rapporti tra la Vdd usata come Vref+ e il risultato della conversione AD (in hex):

Vdd	ADC		Vdd	ADC		Vdd	ADC
5.5	BE		4.2	F9		3.3	13D
5	D1		4.1	FF		3.2	147
4.9	D5		4.0	106		3.1	152
4.8	DA		3.9	10C		3.0	15D
4.7	DF		3.8	113		2.9	169
4.6	E3		3.7	11B		2,8	176
4.5	E9		3.6	123		2.7	184
4.4	EE		3.5	12B		2.6	193
4.3	F3		3.4	134		2.5	1A3

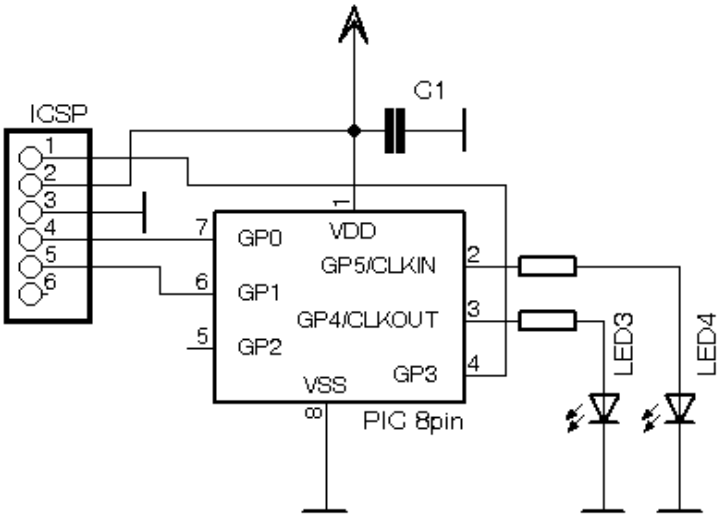
Teoricamente è possibile apprezzare variazioni di 10mV, ma va considerato che l'uscita di FVR è soggetta ad una certa tolleranza e per ottenere valori assoluti precisi si devono mettere in campo operazioni matematiche e tarature, anche se non è questo lo scopo della misurazione.

In ogni caso, utilizzando un foglio di calcolo è facile tracciare una tabella che relazioni la Vdd con il valore che viene ottenuto dalla conversione nella misura della FVR. Con questa tabella si potrà derivare una lookup table per la misura della Vdd o disporre di valori limite a cui devono corrispondere particolari azioni, ad esempio per monitorare lo stato della tensione delle batterie che alimentazione il circuito e segnalare lo situazione di scarica.
Questa misura "inversa", come detto, ha una precisione limitata rispetto a quella ottenibile con un metodo "diretto", ma per l' impiego previsto, è più che adeguata e consente di valutare con

sufficiente cura la situazione della Vdd. Inoltre, rispetto ad una misura "diretta" con il convertitore ADC, presenta notevoli vantaggi:

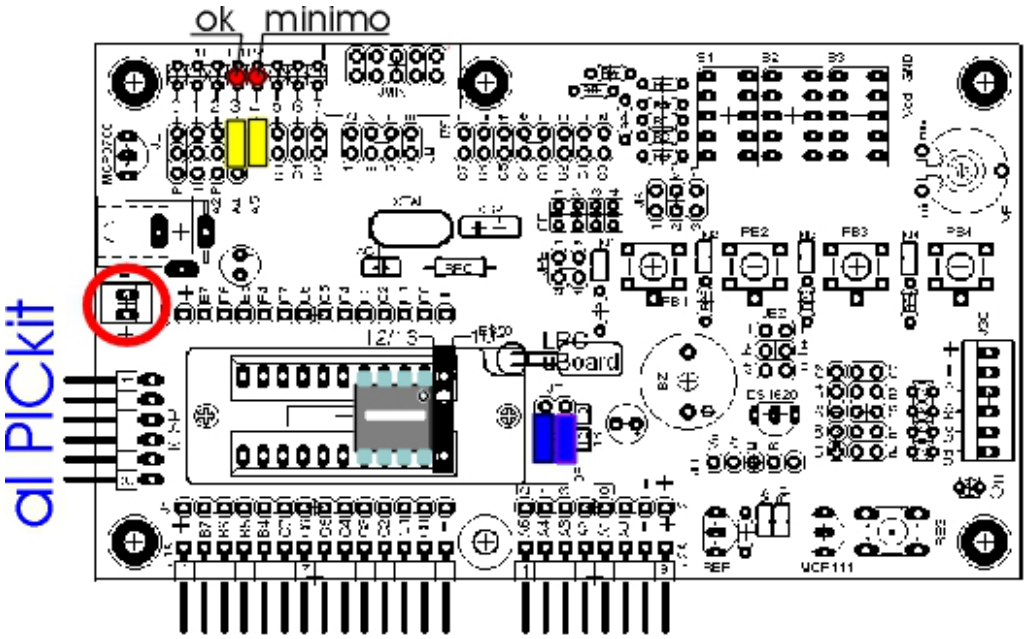
- **non viene utilizzato un pin esterno**, cosa indispensabile per chip con basso numero di pin
- **non richiede un partitore esterno** che, nelle applicazioni low power, è una aggiunta poco gradita
- **non richiede una tensione di riferimento esterna** per la precisione della misura, dato che essa è "intrinseca" al metodo di misura stesso

Lo schema elettrico è evidentemente semplice.



Usiamo due LED: uno acceso quando la tensione è corretta e l'altro che si accende quando la tensione è troppo bassa.

Sulla scheda di sviluppo:



Il sorgente comprende l'inizializzazione del modulo FVR per erogare 1.024 sul buffer indirizzato al

modulo ADC.

```
// configura FVR per 1024 e ADC
    FVRCON = 0b11000001;
```

Per il modulo ADC, impostiamo la Vdd come Vref, il clock di conversione con FRC e selezioniamo l'ingresso FVR. Il risultato della conversione è giustificato a destra per avere disponibili tutti i 10 bit per le successive operazioni.

```
// configura ADC
    ADCON1bits.ADCS = 0b011;        // clock FRC
    ADCON1bits.ADFM = 1;            // right justified
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b11111;       // input da FVR
    ADCON0bits.ADON = 1;            // ADC on
```

Effettuiamo un burst di 4 conversioni e una media semplice come abbiamo visto nell'esempio precedente.

```
 // campiona ingresso ADC
     for(n=0; n<4;n++){
      __delay_us(10);                    // tempo acquisizione
      ADCON0bits.GO = 1;                 // avvia conversione
      while (ADCON0bits.GO);             // attesa fine conversione
      adres16 = (256U *ADRESH |ADRESL); // unisci in 16 bit
       adval = (adval + adres16);        // somma samples
```

Compariamo il risultato della conversione con il valore limite che corrisponde a 3.5V circa. Se siamo al di sopra è acceso un LED, altrimenti viene acceso l'altro.

```
    adval = adval>>2;                    // divide per 4
    if (adval>0x13D){                    // se oltre limite
     alarm=1;                           // allarme on
     ok=0;
    }else{                             // altrimenti
      alarm =0;                        // allarme off
     ok=1;
```

Dato che la tensione della batteria scende in un tempo abbastanza lungo rispetto al ciclo delle istruzioni del processore, per ridurre il consumo possiamo effettuare queste operazioni cadenzate ogni tot secondi; nelle pause il processore è in modalità sleep. Attiviamo il WDT prima di entrare nel loop principale.

```
  //WDTCONbits.WDTPS = 0b01011;      // 2 secondi (default))
    WDTCONbits.SWDTEN = 1;           // WDT on
```

2s è già il valore di default. Se vogliamo applicare un tempo diverso, programmiamo i bit WDTPS. Attiviamo il WDT prima di entrare nel loop principale, cosa possibile perchè abbiamo abilitato nella configurazione iniziale il controllo software del WDT.

```
#pragma config WDTE=SWDTEN   // soft control
```

In questo modo il burst di letture dell'ADC verrà effettuato solo ogni 2s.

Per mettere in opera questa esercitazione **sarebbe utile disporre di un alimentatore variabile**; questo offrirebbe uno spazio maggiore di sperimentazione, con la possibilità di provare diversi valori limite.
In mancanza di questo, **si potranno usare 3 batterie AA o AAA in serie** per alimentare l'hardware. Tre batterie alcaline in serie rendono 4.5V se cariche: la Vdd è maggiore del limite impostato. Se ne togliamo una, la tensione scende a 3V, al di sotto del limite e questo produrrà l'accensione del LED di sotto tensione.

E' indispensabile osservare con cura che la tensione di alimentazione della scheda non superi in alcun caso 5.5V, né tanto meno sia applicata con polarità invertita.

Questi errori, per quanto di brevissima durata, possono danneggiare irrimediabilmente sia il chip che il tools di sviluppo!

E' buona norma, se si usa una alimentazione esterna, disconnettere il PICKIT prima di applicare la tensione.

Per un riferimento completo sul metodo di misura usato, si può consultare **Microchip AN1072**

---

es12C_5

```
/*******************************************************************
*-------------------------------------------------------------------
*
*    Titolo        :  C Enhanced - es12C_5
*                     Sperimentazione su ADC.
*                     Valutazione Vdd con FVR e segnalazione su 2 LED
*                     Temporizzazione letture ADC con WDT
*    Data          :  02-10-2019
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  PIC12F1572
*    Autore        :  afg
*-------------------------------------------------------------------
*******************************************************************
*  Impiego pin :
*  ---------------
*       12F1571/2 @ 8 pin
*
*                  |__\/__|
*             Vdd -|1    8|- Vss
*             RA5 -|2    7|- RA0
*             RA4 -|3    6|- RA1
*        RA3/MCLR -|4    5|- RA2
*                  |______|
*
*    Vdd                1: ++
*    RA5/OSC1/CLKIN     2: Out LED minimo
*    RA4/OSC2/CLKOUT    3: Out LED ok
*    RA3/!MCLR/VPP      4: In  nc
*    RA2                5: An  AN2
*    RA1/ICSPCLK        6: In  nc
*    RA0/ICSPDAT        7: In  nc
*    Vss                8: --
*
* Note: - INTOSC default @ 500kHz*
*******************************************************************/
// WDT = SWDTEN
#include "C:\Corso_C\MyIncludes\conf1572_3.h"

#include <xc.h>

#define alarm LATAbits.LATA5   // LED sotto tensione
#define ok    LATAbits.LATA4   // LED tensione ok


/***** VARIABILI *****/
unsigned int   adval = 0;      // valore media ad
unsigned int   adres16 = 0;
unsigned char  n = 0;

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

/* Inizializza hardware */

    ANSELA = 0;                     // no analogica

    TRISA = 0b00001111;             // RA5:4 uscita
```

```
    LATA = 0;                       // LED spenti

    OPTION_REGbits.nWPUEN = 0;      // abilita WPU

// configura FVR per 1024 e ADC
    FVRCON = 0b11000001;

// configura ADC
    ADCON1bits.ADCS = 0b011;        // clock FRC
    ADCON1bits.ADFM = 1;            // right justified
    ADCON1bits.ADPREF = 0b00;       // Vref+ da Vdd
    ADCON0bits.CHS = 0b11111;       // input da FVR
    ADCON0bits.ADON = 1;            // ADC on

  //WDTCONbits.WDTPS = 0b01011;     // 2 secondi (default))
    WDTCONbits.SWDTEN = 1;          // WDT on

    /*** Main loop ***/
  while (1) {
      SLEEP();

      adval = 0;
    // campiona ingresso ADC
      for(n=0; n<4;n++){
      __delay_us(10);                   // tempo acquisizione
      ADCON0bits.GO = 1;                // avvia conversione
      while (ADCON0bits.GO);            // attesa fine conversione
      adres16 = (256U *ADRESH |ADRESL); // unisci in 16 bit
        adval = (adval + adres16);      // somma samples
      }                           // chiusura for
    adval = adval>>2;                   // divide per 4
    if (adval>0x13D){                   // se oltre limite
       alarm=1;
     ok=0;                              // allarme on
    }else{                              // altrimenti
       alarm =0;
     ok=1;                              // allarme off
    }                           // chiusura if
  }                             // chiusura while
```

Questa versione è fornita come progetto completo (MPLABX v.4.13 o successivi, XC8 v.1.34 o successivi e PIC16F1619.
I file del progetto sono previsti per stare in una cartella ***C:\Corso_C\es12C_5***

Schede informative dettagliate.

Esercitazione 12

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

- **Il modulo ADC**
- **Procedura corretta per il setup del convertitore AD**
- **Orientarsi nelle rappresentazioni numeriche**
- **Convertire in BCD**

Files allegati

Sono allegati i seguenti files/cartelle:

- **Cartella *es12C* : progetti e sorgenti**

Il modulo ADC.

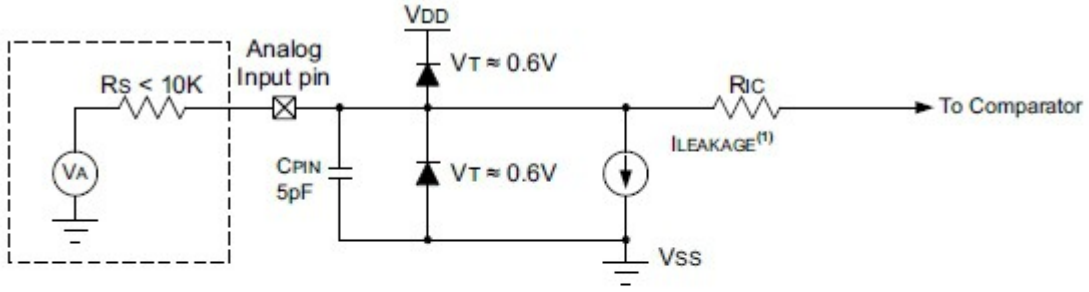
Il mondo esterno al microcontroller è composto da grandezze analogiche. Possiamo ottenere segnali elettrici analogici da temperatura, pressione, umidità, velocità, accelerazione, ecc.
I componenti o circuiti che rendono questi segnali elettrici proporzionali alla grandezza da valutare prendono il nome di sensori.

Però, il microcontroller tratta numeri digitali: occorre convertire il valore analogico in un equivalente digitale : il modulo ADC effettua questa conversione, solitamente con una risoluzione di 8, 10 o 12bit (risoluzioni superiori non ricadono nel campo di impiego diretto del microcontroller, ma sono ottenibili con componenti esterni).

Non trattiamo qui tutti gli aspetti della conversione da analogico a digitale (errori, aliasing, velocità di campionamento, ecc), cosa che richiederebbe un intero volume e, peraltro, non è il tema di queste righe, il cui scopo è descrivere le basi di come impiegare praticamente il modulo ADC.
Chi ha necessità di maggiori informazioni, trova in rete una grande quantità di pagine sull'argomento.

In ogni caso, è opportuno fornire alcune indicazioni di massima sulla conversione AD.

In primo luogo, il convertitore necessita all'ingresso di una tensione continua, variabile o meno, ma entro determinati limiti. In particolare, l'ingresso di un canale analogico può essere rappresentato da questo diagramma.



Legend: CPIN = Input Capacitance
ILEAKAGE = Leakage Current at the pin due to various junctions
RIC = Interconnect Resistance
RS = Source Impedance
VA = Analog Voltage
VT = Threshold Voltage

Osserviamo che sono presenti due diodi di protezione che impediscono alla tensione di ingresso di di superare ***Vdd+Vt*** o ***Vss-Vt*** , andando in conduzione e scaricando la corrente sui rami dell'alimentazione. Per una alimentazione di 5V, ad esempio, non si potranno superare i limiti -0.5V / +5.5V.

Però questi diodi hanno una bassissima capacità di dissipazione e se la corrente all'ingresso non è limitata si distruggono rapidamente.

Quindi:

il modulo ADC del microcontroller può trattare in ingresso segnali compresi tra Vss e Vdd. L'applicazione di tensioni inferiori alla Vss o superiori alla Vdd danneggiano il chip in modo definitivo.

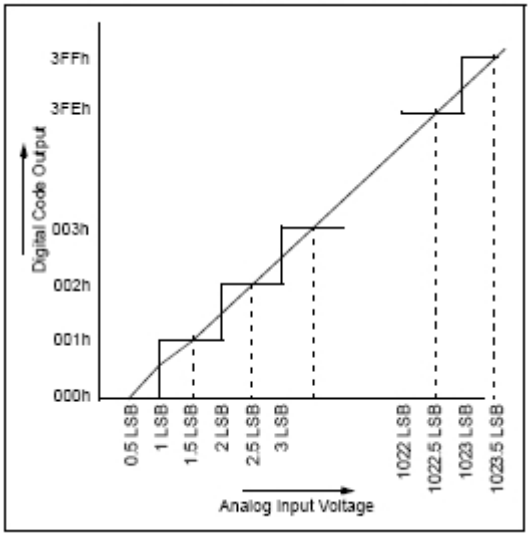
Se abbiamo la necessità di convertire questo genere di segnali dovremo provvedere ad un circuito di condizionamento esterno.

In secondo luogo, la conversione tra la grandezza analogica, continua, e quella digitale, discontinua, avviene con **una certa approssimazione**.
Questo vuol dire che il valore digitale in uscita dal convertitore non corrisponderà ad un valore univoco della grandezza continua, ma ad un certo intervallo di valori.

Si definisce **risoluzione** il numero di bit massimo in uscita dal convertitore. Da questo dipende la "finezza" della conversione. Ad esempio, disponendo di un convertitore a 8 bit, una tensione di ingresso da 0 a 5V sarà valutata in un massimo di 256 frazioni, grandi ognuna 195.31mV.
Questo vuol dire che, se una tensione di 197 o 220mv otterremo dal convertitore lo stesso valore binario; non sarà possibile definire valori in ingresso minori della risoluzione.

Per ottenere un maggiore risoluzione occorrerà utilizzare un convertitore con un numero maggiore di bit: ad esempio, un AD a 10 bit (1024) permette, nel caso esemplificato prima, di discriminare passi di 4.88mV.

FIGURE 19-2: A/D TRANSFER FUNCTION



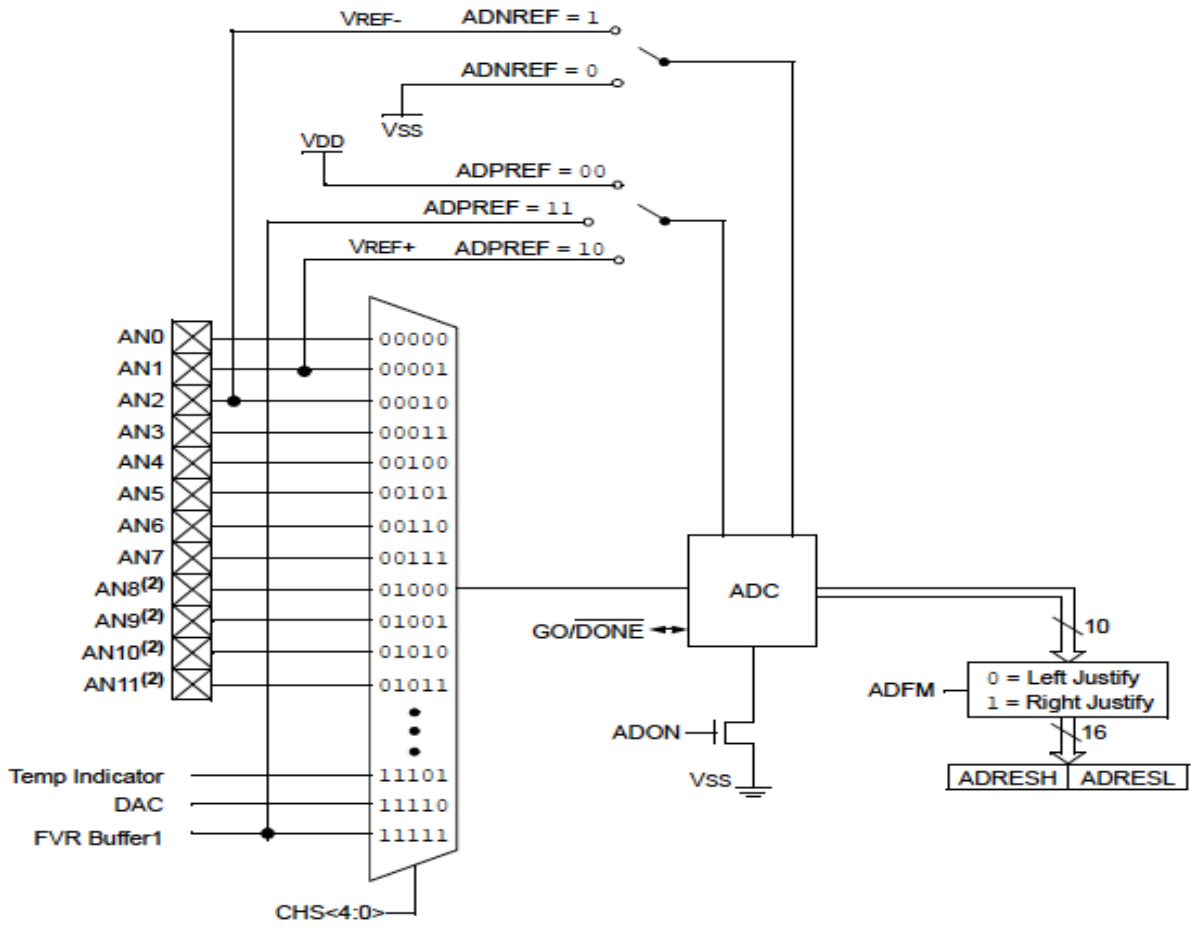
In sostanza, l'uscita del convertitore è una rampa a gradini, in numero pari alla risoluzione.

La lunghezza di uno dei gradino si definisce anche quanto e identifica la minima variazione della tensione in ingresso che comporta un diverso valore digitale in uscita.

La tensione massima di fondo scala è chiamata tensione di riferimento e, in generale, può essere semplicemente la tensione di alimentazione oppure provenire da una sorgente interna o esterna di tensione.

Va notato che gli step vanno da 0 a 1023 ed ognuno è ampio un lsb

Il tipico modulo ADC ha questa struttura

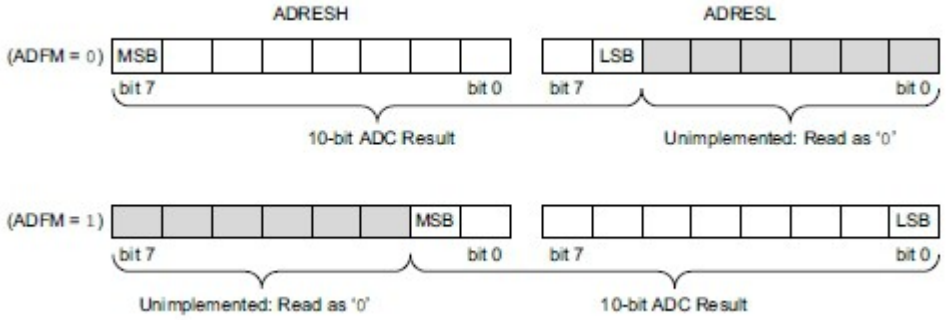


- sono disponibili vari ingressi analogici (detti canali) al modulo ADC, in relazione al numero dei pin disponibili per quel determinato chip. La scelta del canale da convertire è effettuata da un multiplex comandato dai bit di un registro di controllo.
  Per chi è rimasto ai PIC Midrange, negli Enhanced i canali da convertire sono selezionabili singolarmente senza limitazioni o combinazioni obbligate.
- sono accessibili anche canali interni: l'uscita del DAC, il buffer FVR e il sensore di temperatura, il che permette alcune interessanti applicazioni.
- la tensione di riferimento della conversione, nel caso più semplice, è la tensione di alimentazione (Vdd-Vss), ma si possono applicare tensioni di riferimento interne (FVR) o esterne (pin Vref+/Vref-)
- Il convertitore vero e proprio è del tipo ad approssimazioni successive

Non tutti i moduli ADC hanno le stesse opzioni: ad esempio, nei più semplici, il riferimento negativo non è selezionabile, ma è sempre Vss, ecc. Quindi è opportuno verificare sul foglio dati del componente usato quali sono le sue specifiche possibilità.

Se il risultato della conversione è su 8 bit, basterà un byte per contenerlo. Se la risoluzione è a 10 o 12 o14 bit, occorrono due bytes.

In questi casi sono possibili due disposizioni del risultato, selezionabili con un bit di un registro di controllo. Vediamo un esempio per un ADC a 10bit:



Nel primo caso, abbiamo un risultato giustificato a sinistra (left justified) : i bit più significati deol risultato della conversione sono raccolti nel byte **ADRESH**, mentre nel byte **ADRESL** troviamo i du bit meno significativi.
Nel secondo caso abbiamo una giustificazione a destra (right justified), con i due bit più significativi in **ADRESH** e tutti gli altri in **ADRESL**.

Non si tratta di due opzioni bislacche, ma **di scelte funzionali all'applicazione.**

La giustificazione a sinistra rende una conversione effettiva di 8 bit, poiché i due bit meno significativi si trovano in **ADRESL**. Basterà leggere **ADRESH**. Questo utile quando si desidera un risultato non al massimo della precisione e che è possibile trattare direttamente scon registri a 8 bit. L'eliminazione dei due bit meno significativi può essere una via per eliminare disturbi o piccole variazioni indesiderate del segnale di ingresso che renderebbero incerta la conversione.
In questa modalità, l'intervallo ADC effettivo è: **ADRESH = 0 @ 0V -> 255 @ Vref.**

La giustificazione a destra fornisce il risultato a 10 bit completo (o 12 bit per i chip con ADC 12bit) con spazio sufficiente nel risultato per consentire all'utente di aggiungere letture per una media senza overflow. Per contro, si devono gestire risultati su due bytes e quindi operazioni matematiche e comparazioni più complesse.

Da un punto di vista del codice, in C il risultato giustificato a destra sarà un **int**:

```
unsigned int adcresult;              // 16bit

adcresult = ADRESH*256U + ADRESL;
```
oppure
```
adcresult = (ADRESH <<8) + ADRESL;
```

Mentre se abbiamo scelto la giustificazione a sinistra, **ADRESH** sarà trattato come un singolo char.

Non potendo trattare qui la teoria del funzionamento di un convertitore AD, dobbiamo però dire che la tipologia presente nei PIC richiede un clock per il suo funzionamento.

Questo clock può essere derivato da due fonti:

- il **clock principale**
- l'**oscillatore interno FRC**

Il clock interno consente di ottenere le conversioni più rapide., anche se esiste un limite,

generalmente attorno a 1us per bit. Una conversione a 10 bit richiede un tempo di 11.5us al minimo per essere completata, più il tempo di acquisizione, per cui 20us circa.

Ovviamente non tutte le frequenze possibili del clock primario sono adeguate, per cui sono disponibili divisori con fattori da 2 a 64.

Va osservato che il clock principale è sospeso in modalità sleep e quindi, se si usa questo, il modulo ADC non è operativo.
Per contro, usando il clock **FRC**, che ha una valore non preciso e stabile come il clock principale, si è sicuri di rientrare sempre nelle corrette specifiche del tempo di conversione qualsiasi sia il clock primario. Inoltre **FRC** è operativo anche in sleep e in questo caso il modulo ADC può diventare una sorgente di wake-up.

Il modulo ADC è anche una sorgente di interruzione al completamento della conversione.

La gestione del convertitore ADC comprende le seguenti azioni:

- configurazione dei pin come ingressi analogici (default)
- selezione della tensione di riferimento
- selezione del clock della conversione
- selezione del formato del risultato
- scelta dell'operatività in interrupt o senza
- selezione del canale di ingresso
- avvio della conversione
- attesa di fine conversione

Per gestire tutte queste opzioni, il modulo ADC dispone di più registri di controllo.
Il primo è **ADCON0**. Nell'immagine seguente un esempio per ADC con11 canali.

U-0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
—	CHS4	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

I bit **CHS<4:0>** permetto di selezionare il canale da convertire.
La codifica è abbastanza semplice:

```
CHS<4:0>= 0b00000 -> AN0
CHS<4:0>= 0b00001 -> AN1
CHS<4:0>= 0b00010 -> AN2
```

ecc.
Sono disponibili molte opzioni per la scelta dei canali, ma saranno utilizzabili solo quelle relative alle possibilità del chip. In generale, chip con più pin avranno più canali AD.
In ogni caso, gli ultimi tre valori corrispondono alle sorgenti di tensione interne:

```
CHS<4:0>= 0b11101 -> indicatore di temperatura
CHS<4:0>= 0b11110 -> uscita del DAC
CHS<4:0>= 0b11111 -> buffer1 di FVR
```

Troviamo, poi, un bit **ADON** che, portato a 1, abilita il convertitore. Il default è 0, quindi

all'accensione l'ADC è spento.

Attenzione: anche se il modulo ADC è spento, i pin al default assumo la funzione di ingressi analogici in ogni caso. Questo impedisce l'uso della funzione digitale.

Per disabilitare gli ingressi analogici sarà necessario portare a 0 i relativi bit dei registri `ANSELx`.

Necessario un cenno particolare al bit2 **`GO_nDONE`**.
Questo bit ha una doppia funzione:

- **va scritto a 1 per avviare la conversione** e resta a 1 fino a che la conversione è in corso
- **ritorna a 0 automaticamente quando la conversione è completata**

Quindi, oltre che il "pulsante di avvio" della conversione, è anche il flag che ne indica la fine.

Da osservare che il convertitore deve essere avviato con una istruzione separata da quella che configura gli altri bit del registro `ADCON0`.

Va notato che il bit **`GO_nDONE`**, pur essendo un indicatore di fine conversione, non è il flag di fine conversione dell'interruzione, che è **`ADIF`** nel registro **`PIR1`**. Questo flag va a 1 a fine conversione e va resettato da programma quando si utilizza l'interrupt.
Come immaginabile, c'è un bit **`ADIE`** nel registro **`PIE1`** per abilitare la richiesta di interruzione.
ADC è un modulo periferico, per cui l'interrupt richiede il settaggio sia di **`PEIE`** che di **`GIE`.**

Il secondo registro di controllo è **`ADCON1`**:

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	U-0	U-0	R/W-0/0	R/W-0/0
ADFM	ADCS<2:0>			—	—	ADPREF<1:0>	
bit 7							bit 0

Con questo registro possiamo programmare:

- il clock della conversione
- la giustificazione del risultato
- la tensione di riferimento

Il bit **ADFM** a 0 rende la giustificazione a destra (default).
Se portato a 0 rende la giustificazione a sinistra.

I bit **`ADPREF`** consentono di variare la tensione di riferimento positiva:
`ANPREF<1:0>  00  Vdd` `(default)`

```
01  riservato
10  pin Vref+
11  FVR buffer1
```

Va da se che la tensione di riferimento dovrà essere maggiore o uguale a quella da convertire.
Alcuni ADC dispongono anche dei bit 3:2 che diventano **ADNREF** per scegliere la sorgente del riferimento negativo.
Dove questi bit mancano, come nel caso in esempio, il riferimento negativo è fisso a Vss.

La formula della quantificazione della conversione può essere generalizzata in:

Vref+ - Vref- / bit_risoluzione

Ad esempio, un convertitore con risoluzione a 10bit (1024 passi) e una tensione di riferimento tra Vss e FVR = 4.096V avrà una definizione di 4mV per step. Così, se il risultato della conversione è 0x1FF (511 decimale) la tensione sarà 4mv * 511 = 2.044V.

Una analisi più dettagliata è richiesta dai bit **ADCS<2:0>**. Attraverso questi è possibile impostare il clock della conversione.
La funzione è questa:

```
bit 6-4 ADCS<2:0>: ADC Conversion Clock Select bits
        111 = FRC - RC interno
        110 = FOSC/64
        101 = FOSC/16
        100 = FOSC/4
        011 = FRC - RC interno
        010 = FOSC/32
        001 = FOSC/8
        000 = FOSC/2
```

La procedura di conversione richiede due tempi:

1. è necessario un primo tempo di acquisizione del segnale ***Tacq***. Una volta applicata la tensione da convertire al pin di ingresso, occorre una attesa che permetta la carica di un condensatore (***CHOLD***) di **sample/hold** interno al chip.

Questo è dovuto al meccanismo del convertitore. Normalmente il condensatore CHOLD è collegato al pin di ingresso. Quando portiamo il bot GO a 1, questo condensatore viene staccato dal pin e collegato al convertitore: che la conversione sarà effettuata sulla tensione del condensatore.
Questo evita che il segnale campionato possa variare durante la conversione, falsando il risultato.
Il condensatore è riconnesso al pin alla fine della conversione.

La carica del condensatore richiede che l'impedenza della sorgente sia minore di 10kohm ed è in base a questo valore che si effettuano i calcoli. Se l'impedenza è maggiore, occorrerà adeguare di conseguenza il tempo di acquisizione. Se è minore, il tempo di acquisizione potrebbe essere minore.

Un esempio di calcolo del tempo di acquisizione, che dipende anche dalla temperatura:

Con una impedenza della sorgente di 10k e una temperatura di 50°C:

Tacd = amplifier settling time + CHOLD charging time + temperature coefficient =
= Tamp + Tc+Tcoff = 2us +Tc+[(temperature-25°)(0.05us/°C)]C

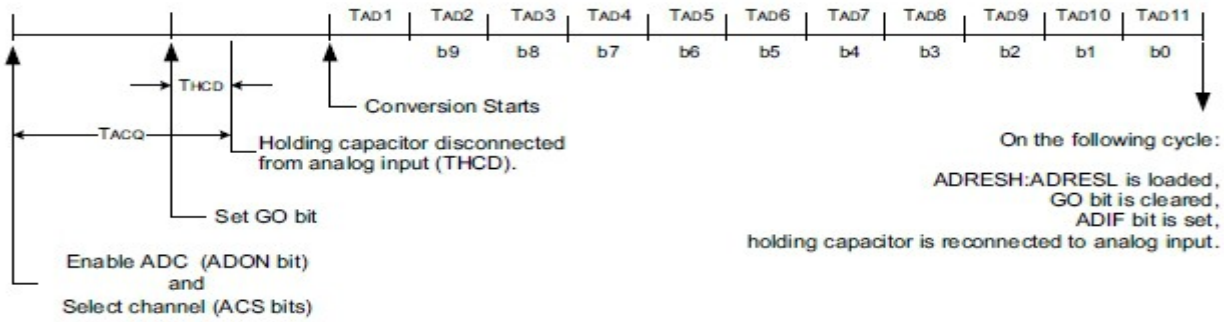
valutando anche gli altri parametri necessari (formule complete sul foglio dati) abbiamo un tempo minimo di acquisizione di 4.37us, che, solitamente, viene portato a 10us per sicurezza,.

2. Quando abbiamo definito il tempo di acquisizione, possiamo avviare la conversione. Dipendendo da un clock per avanzare nei bit della risoluzione, è evidente che maggiore sarà la frequenza del clock, minore sarà il tempo necessario a completare la conversione. Questo aspetto assume una particolare importanza quando si devono campionare segnali variabili, con tracce audio video.

Il problema è che non tutte le frequenze di clock sono adeguate alla conversione.
Il foglio dati riporta una tabella con le possibili combinazioni.

ADC Clock Period (TAD)		Device Frequency (Fosc)				
ADC Clock Source	ADCS<2:0>	20 MHz	16 MHz	8 MHz	4 MHz	1 MHz
Fosc/2	000	100 ns	125 ns	250 ns	500 ns	2.0 µs
Fosc/4	100	200 ns	250 ns	500 ns	1.0 µs	4.0 µs
Fosc/8	001	400 ns	500 ns	1.0 µs	2.0 µs	8.0 µs
Fosc/16	101	800 ns	1.0 µs	2.0 µs	4.0 µs	16.0 µs
Fosc/32	010	1.6 µs	2.0 µs	4.0 µs	8.0 µs	32.0 µs
Fosc/64	110	3.2 µs	4.0 µs	8.0 µs	16.0 µs	64.0 µs
FRC	x11	1.0-6.0 µs	1.0-6.0 µs	1.0-6.0 µs	1.0-6.0 µs	1.0-6.0 µs

Il tempo **Tad** indica il periodo per la conversione di un bit.
Le caselle evidenziate indicano combinazioni **al di fuori** delle possibilità del convertitore.
In pratica, sono adeguati tempi **Tad** da 1 a 6us.
Siccome la conversione completa a 10bit impegna **11.5Tad** in base a questo schema:



ne risulta che il minimo tempo possibile sarà di **11.5us + Tacq**,

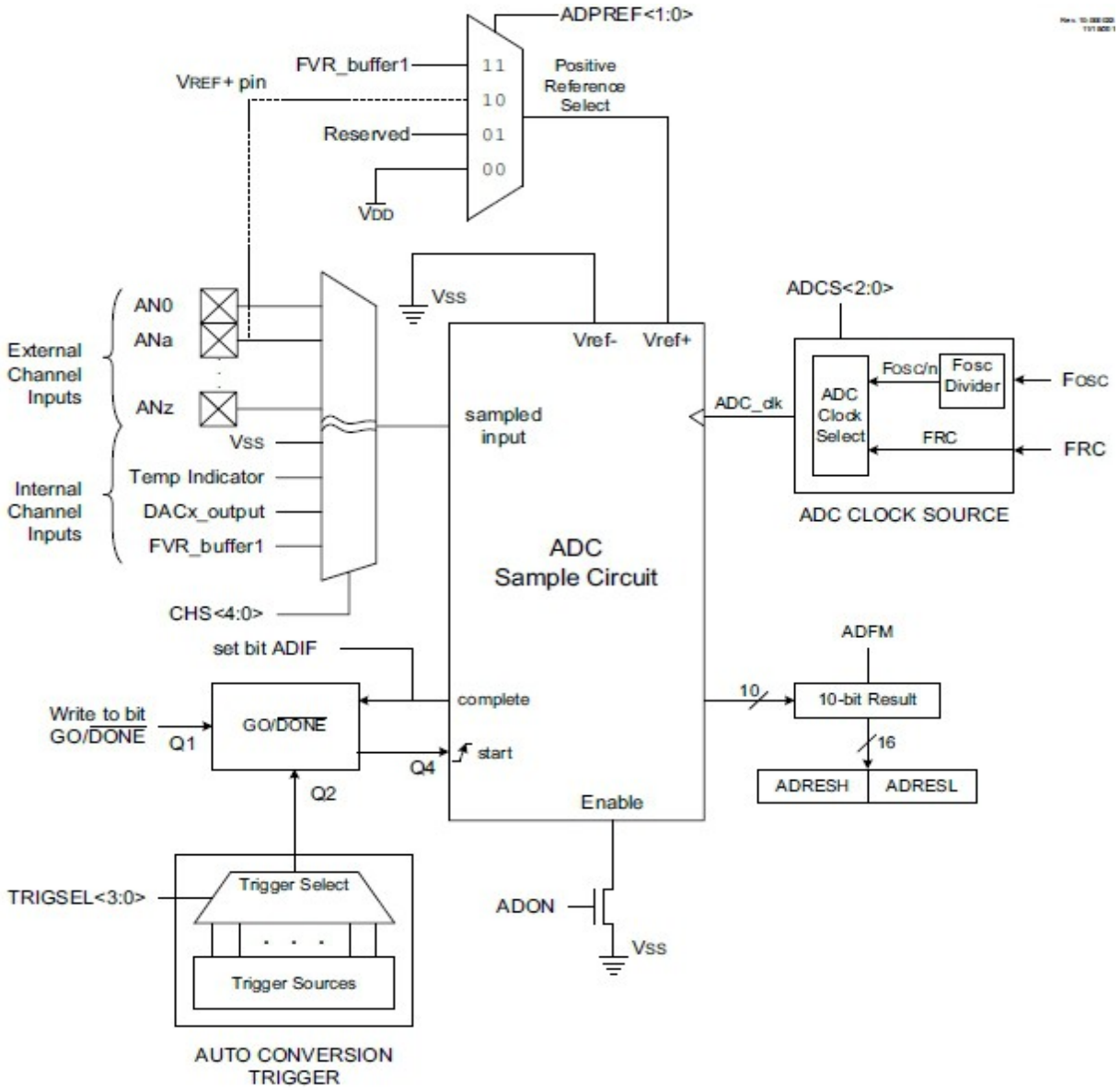
Se utilizziamo il **clock FRC**, abbiamo un **tempo medio di TAD=1.7us** (ma che, per temperatura,

tensione e parametri costruttivi, può variare tra 1 e 6us).
Va considerato che il foglio dati suggerisce di utilizzare FRC se il clock del sistema supera 1MHz, ovvero la correttezza del risultato della conversione si riduce con l'aumentare della frequenza.

Va tenuto presente che il clock può variare, anche durante il funzionamento del programma, se richiesto, e questo necessita il controllo del tempo di conversione adeguato. Per contro, usando FRC il tempo di conversione è lo stesso per qualsiasi clock primario.

Alcuni chip dispongono di moduli ADC con un maggior numero di opzioni, ad esempio quella di avere un trigger che avvia la conversione in modo del tutto automatico.

Vediamo qui il diagramma funzionale dell'ADC del PIC16F1619.



Un blocco denominato **Auto Conversion Trigger** consente di avviare la conversione (allo stesso modo ottenuto "manualmente" dal programma agendo sul bit **`GO_nDONE`**) quando avviene un

evento su una altra periferica. Questo consente una notevole automatizzazione del processo di acquisizione ed una riduzione delle istruzioni del sorgente.

Nel caso in esempio, una tabella riporta le possibili sorgenti del trigger:

TABLE 17-2: AUTO-CONVERSION SOURCES

Source Peripheral	Signal Name
Timer0	T0_overflow
Timer1	T1_overflow
Timer2	TMR2_postscaled
Timer4	TMR4_postscaled
Timer6	TMR6_postscaled
Comparator C1	C1_OUT_sync
Comparator C2	C2_OUT_sync
SMT1	SMT1_CPW
SMT1	SMT1_CPR
SMT1	SMT1_PR
SMT2	SMT2_CPW
SMT2	SMT2_CPR
SMT2	SMT2_PR
CCP1	CCP1_out
CCP2	CCP2_out

Altri chip dispongono di una versione avanzata del modulo ADC, chiamata ADC2, che offre varie opzioni di calcolo dei risultati della conversione.
Uno sguardo ai fogli dati dei vari componenti permette di avere maggiori informazioni.

La corretta procedura di setup del convertitore AD

Per riassumere, la **corretta procedura di setup del convertitore AD, secondo la documentazione di Microchip, è la seguente**:

1. abilitare il pin di ingresso analogico come input nel **`TRIS`** (default)
2. abilitare la funzione analogica con **`ANSEL`** (default)
3. **disabilitare il weak pull-up del pin** (**`WPUx`** e **`OPTION_REG`**)
4. selezionare il clock della conversione
5. selezionare la tensione di riferimento
6. scegliere il canale da convertire
7. scegliere la giustificazione del risultato
8. accendere il modulo
9. configurare l'interrupt (dove richiesto)
10. cancellare il flag di interrupt
11. abilitare l'interrupt dell'ADC (**`ADCIE`**)
12. abilitare l'interrupt periferico (**`PEIE`**)
13. abilitare l'interrupt globale (**`GIE`**)
14. attendere il tempo si acquisizione.
15. avviare la conversione (**`GO_nDONE =1`**).
16. attendere la fine della conversione
    - in polling (**`GO_nDONE =0`**)
    - oppure con l'interrupt
17. cancellare il flag di interrupt (**`ADIF`**)
18. leggere il risultato nei bytes **`ADRESH:L`**

Orientarsi nelle rappresentazioni numeriche.

Per poter sfruttare al meglio le possibilità dei sistemi digitali occorre avere una sufficiente dimestichezza con le diverse forma di rappresentazione dei numeri:

- binario
- esadecimale
- decimale - BCD

Ci sarebbe anche il sistema ottale, ma, nella pratica, viene usato assai raramente e possiamo trascurarlo del tutto.

Non facciamo qui una trattazione teorica dei sistemi numerici, per la quale sono disponibili numerosissime fonti, ma ci limitiamo a cercare di fornire il minimo indispensabile per potersi orientare.

In primo luogo:

dovrebbe diventare chiaro che, **all'interno di un computer o di un microcontroller, i numeri esistono solo nella rappresentazione binaria.**

Anzi, **all'interno del computer NON esiste alcun numero**: i dati sono immagazzinati in celle di memoria che contengono dei livelli di tensione vicini alla alimentazione e che chiamiamo livello alto, oppure vicini alla massa e che chiamiamo livello basso. E nient' altro.

Solo se attribuiamo al livello alto il valore 1 e al livello basso il valore 0, cominciamo a poter utilizzare questi all'interno di una logica derivata dal lavoro di **Boole** (www) (logica booleana) dove le rappresentazioni di numeri e simboli sono effettuate con le due cifre binarie 0 e 1.
Se raccogliamo queste celle di memoria (che sono i bit dell'informazione) in gruppi da 8 (8 bit = 1 byte) o da 16 (= 1 word) o maggiori, ed avendone a sufficienza, possiamo rappresentare con questi ogni genere di numero o simbolo ed agire su di essi con operatori logici e aritmetici.

Dobbiamo ribadire che, per la macchina (per ora), non esiste alcun concetto di numero e neppure di operazione logica o aritmetica: si tratta solamente di azioni meccanicamente predefinite su questi gruppi di valori di tensione alto/basso secondo le regole imposte da come sono realizzati i circuiti (ALU – unità logica aritmetica) e le “istruzioni” che compongono i programmai.
Siamo NOI che attribuiamo a questi insiemi di livelli logici il valore di numeri, lettere e altro.

La rappresentazione classica vede il bit meno significativo (lsb) a destra e il più significativo (msb) a sinistra.

	msb							**lsb**
bit	7	6	5	4	3	2	1	0

In questo non c'è nulla di strano , né di diverso dal sistema decimale che dovremmo conoscere: in un numero decimale le cifre hanno lo stesso aspetto. Il valore dipende dalla posizione (sistema posizionale), con l'estrema destra del numero in cui il valore è unitario e sale di *10 ad ogni spostamento a sinistra.

Così, ad esempio, il numero decimale 12019 va letto da destra a sinistra:

colonna	4	3	2	1	0
cifra	1	2	0	1	9
valore	* 10000	* 1000	* 100	* 10	* 1

Quindi 12019 vuol dire:

- 9 volte 1 +
- 1 volta 10 +
- 0 volte 100 +
- 2 volte 1000 +
- 1 volta 10000

Se lo vediamo sotto forma di potenze di 10, abbiamo:

colonna	4	3	2	1	0
cifra	1	2	0	1	9
esponente di 10	* 10^4	* 10^3	* 10^2	* 10^1	* 10^0

Dal quale vediamo come **il numerare a partire da 0 e non da 1 la prima colonna a destra sia giustificato.**

Del tutto similmente, ad esempio, il numero binario 11001

colonna	4	3	2	1	0
cifra	1	1	0	0	1
valore	* 16	* 8	* 4	* 2	* 1
esponente di 2	*2^4	*2^3	*2^2	*2^1	*2^0

vuol dire:

- 1 volte 1 +
- 0 volte 2 +
- 0 volte 4 +
- 1 volte 8 +
- 1 volta 16 = 25 decimale

Chi ha difficoltà a queste conversione può utilmente impiegare la Calcolatrice di Windows che ha l'opzione di trattare numeri esadecimali, binari, decimali e ottali.

Nel microcontroller a 8 bit, la lunghezza base delle cifre del numero binario è 8 (8 bit = 1 byte), il che permette di contenere numeri binari da 00000000 a 11111111 ovvero da 0 a 255 decimale.

bit	7	6	5	4	3	2	1	0
cifra	1	1	1	1	1	1	1	1
valore	*128	*64	*32	* 16	* 8	* 4	* 2	* 1
esponente di 2	$*2^7$	$*2^6$	$*2^5$	$*2^4$	$*2^3$	$*2^2$	$*2^1$	$*2^0$

Quindi 11111111 binario equivale a :

1*128 + 1 *64 +132 +1*16 + 1*8 +1*4 + 1*2 +1*1 = 255 decimale

Al di fuori della macchina, in cui rappresentiamo i numeri con livelli logici descritti dal sistema binario, l'uso di numeri binari non è per niente pratico. La nostra mente è un "calcolatore" analogico e non digitale: non usa zero e uno se non quando vuole ragionare con la logica booleana.

Si è fatto, quindi, un passo ulteriore, introducendo i numeri esadecimali.
Se i numeri binari, come abbiamo visto, in fondo sono abbastanza "reali", dato che corrispondono ai livelli logici delle celle di memoria, i numeri esadecimali sono molto più astratti.

Per capirne il senso riprendiamo il nostro byte di 8 bit che possono valere ognuno 1 o 0.

	msb							**lsb**
bit	7	6	5	4	3	2	1	0
nibble	nibble alto				nibble basso			

Possiamo pensare il numero binario diviso in due metà:

- i bit 7:4, più significativi, raggruppati in un blocco che chiamiamo nibble alto
- i bit meno significativi 3:0 che raggruppiamo in un nibble basso

Ogni nibble può valere al massimo 1111, ovvero 15.

Se usiamo una forma numerica che ha come base 16, ecco che abbiamo l'esadecimale o hex.
Con una base binaria, abbiamo la possibilità di usare 2 cifre: 0 e 1
Con una base decimale possiamo usare 10 cifre, da 0 a 9
Con una base esadecimale abbiamo disponibili 16 cifre, per le quali si usano le cifre da 0 a 9, più le lettere A,B,C,D,E,F.

Il byte diviso in due **nibble**, può essere descritto con 8 cifre binarie o due cifre esadecimali.

	msb							**lsb**
bit	7	6	5	4	3	2	1	0
binario	1	1	0	1	0	1	0	1
hex	D				5			

L'uso dell'esadecimale ci semplifica le cose quando trattiamo di quantità di bit. Poche cifre esadecimali possono esprimere numeri binari enormemente grandi, ma anche numeri decimali

grandi, dato che la base 16 è maggiore di 10.
65535, 5 cifre decimali, è espresso con sole 4 cifre esadecimali FFFF e richiede due bytes binari.

Da queste considerazioni, però, non deve sfuggire mai la nozione che nelle celle della memoria ci sono solo, sempre e comunque valori binari.

Se scriviamo la linea

```
unsigned char  x = 128;
```

nella locazione di memoria **x** NON troviamo 128, ma una serie di 8 livelli di tensione logici che possiamo descrivere con il numero binario 10000000.

Così, se scriviamo

```
unsigned char x=0x80
```

non c'è nessun “80”, ma sempre un byte di memoria il cui contenuto è identico al precedente.

Binari ed esadecimali sono utili quando si ha che fare con bit e byte, molto più pratici che non la numerazione decimale.

Se vogliamo portare a 1 il bit 7 e il bit 0 di PORTA, non è difficile veder come un numero binario descriva visivamente la situazione dei livelli logici dei pin del port:

```
PORTA = 0b10000001;
```

mentre:

```
PORTA = 129;
```

non ci dice molto. Per avere una immagine, dobbiamo convertirlo in binario.

Anche se il C in senso stretto si ostina a non contemplare l'uso dei numeri in binario (che non hanno alcun interesse nei grossi sistemi per cui il linguaggio è stato originariamente pensato), le implementazioni pratiche, principalmente quelle dirette alla programmazione dei microcontroller, integrano comunque anche le notazioni binarie, indispensabili in questo ambito.

Nel trattamento di bit in registri, cosa del tutto ordinaria in un microcontroller, il numero binario, come abbiamo visto, è una immediata immagine dei livelli logici da attribuire ad ogni bit.

In generale, quindi, un compilatore come XC8 ci permette di usare tutti e quattro i modi per rappresentare i numeri e questo può trarre in inganno se non si presta la giusta attenzione.
Ad esempio, posso caricare una locazione di memoria con un numero decimale:

```
x = 128;
y = 12;
```

e poi posso sommare i due numeri:

```
z = x + y;
```

ma se voglio presentare su un display il valore di **z**, devo tenere presente che il contenuto della locazione **z** è **10001100** e non 140. E' semplice è ottenere l'equivalente in esadecimale:

	msb							lsb
bit	7	6	5	4	3	2	1	0
binario	1	0	0	0	1	1	0	0
hex	8				C			

Più complesso è ottenere valori decimali.
In effetti si tratta di ottenere valori binari che non superino il numero 9.

In pratica, ogni cifra di un numero è rappresentata da un codice binario di quattro bit, il valore del quale è compreso tra 0 (0000) e 9 (1001).
Ad esempio, il numero 9 in un byte sarà:

	msb							lsb
bit	7	6	5	4	3	2	1	0
binario	0	0	0	0	1	0	0	1
BCD	0				9			

BCD – bynary coded decimal – è una codifica iniziata da IBM e portata avanti in diverse forme. Qui intendiamo **SBCD** o simple BCD, basato su una codifica fino a 4 bit per cifre da 0 a 9, ovvero le cifre possibili nel conteggio decimale.

Se ci limitiamo ad usare solo una metà del byte, l'altra metà viene “sprecata” dato che contiene solo 0 (oppure 1 come nel codice EBCDIC).
Non si tratta però di una forma non utilizzabile, in quanto basta modificare il nibble alto per ottenere caratteri ASCII da inviare in stampa. Ad esempio, un OR con 0x30 trasforma il valore in 39 che è la codifica ASCII per la cifra 9.
Nel byte avremo:

	msb							lsb
bit	7	6	5	4	3	2	1	0
binario	0	0	1	1	1	0	0	1
BCD	3				9			

Se si tratta di immagazzinare numeri, è preferibile un consumo minore di memoria minore. In tal caso possiamo impacchettare due numeri decimali in un solo byte, che conterrà al massimo il valore 99 (BCD packet).

	msb							lsb
bit	7	6	5	4	3	2	1	0
binario	1	0	0	1	1	0	0	1
BCD	9				9			

Rispetto alla possibilità dell'uso delle cifre da 0 a F abbiamo comunque un sotto impiego dello spazio: se dovessimo immagazzinare il numero 99, in binario-esadecimale avremmo

	msb							lsb
bit	7	6	5	4	3	2	1	0
binario	0	1	1	0	0	0	1	1
hex	6				3			

ma con la forma in BCD abbiamo il vantaggio di avere il numero in cifre decimali immediatamente utilizzabili per inviarle a dispositivi come display, stampanti, monitor, linee seriali, magari dopo un rapido spacchettamento e traslazione in singoli caratteri ASCII.

Convertire da binario a decimale.

Il risultato del modulo ADC del microntroller è un numero binario a 8/10/12 bit, ma a noi solitamente serve nella forma decimale o ASCII per la presentazione su un display o la stampa.

Una semplice via per convertire un numero da binario a decimale è quella di utilizzare una serie di istruzioni utilizzando l'operatore modulo %.

c = a % b ;

rende c uguale al resto della divisione di a per b.
Ad esempio:

c = 50 % 10;

rende c = 0 dato che 50/10 = 5 senza resto, mentre

c = 50 % 6;

rende c = 2 dato che 50/6= 8 con resto di due.

Supponiamo di avere il risultato **`adval`** di una conversione analogico-digitale a 8 bit =1 byte (ma anche su due bytes) che corrisponde al binario 10010011 , ovvero all'hex 93 e al decimale 147.
Per passare a decimale dividiamo il numero per 100 (al massimo in un byte ci sta 255) ed usiamo questo risultato per le centinaia.
Dividiamolo per 10 ed usiamo il risultato per le decine. Il resto sono le unità.

147 / 100 = 1 con resto 47 → centinaia = 1
47 / 10 = 4 con resto 7 → decine = 4
→ unità = 7

Questo è quello che appare nella nostra figurazione in decimale. Nel reale abbiamo:

10010011 / 01100100 = 00000001 con resto 00101111 → centinaia = 00000001
00101111 / 00001010 = 00000100 con resto 00000111 → decine = 00000100
→ unità = 00000111

Nella lookup table che converte il valore numerico in maschera per i segmenti del display, abbiamo 10 righe, ognuna indirizzata da un indice che varia tra 0 e 9, ovvero tra 00000000 e 00001001.
Quindi per il valore centinaia avremo indirizzata la prima riga. Per il valore decine avremo indirizzata la quarta riga e per il valore unità la settima riga.

Dalla tabella verranno prese la maschera che accende i segmenti che rappresenta la cifra 1, poi quelli che rappresentano la cifra 4 ed alla fine quelli della cifra 7: sul display leggeremo 147, numero che in questa forma non ha mai trovato posto nei bytes della memoria, dato che la massima cifra per ogni bit è 1.

In C la codifica sarà questa:

```
unit = (adval %10);
dec  = (adval/10)%10;
cent = (adval/100)%10;
```

Questo scompone il numero di partenza in tre bytes ognuno dei quali contiene nel nibble basso il valore decimale.
Ad esempio:

```
adval = 128 (10000000)
cent  = 1   (00000001)
dec   =  2  (00000010)
unit  =   8 (00001000)
```

Se abbiamo un risultato a 10bit il numero massimo rappresentabile sarà 1023; con una conversione a 12 bit avremo un massimo di 4096.
Per convertire interamente questi numeri a 10 o 12 bit, consideriamo di modificare l'algoritmo visto sopra aggiungendo le migliaia.

```
unit = (adval %10);
dec  = (adval/10)%10;
cent = (adval/100)%10;
mill = (adval/1000)%10;
```

Questo scompone il numero di partenza in 4 bytes ognuno dei quali contiene nel nibble basso il valore decimale.
Ad esempio:

```
adval = 1023 (0000001111111111)
mill  = 1    (00000001)
cent  =  0   (00000000)
dec   =   2  (00000010)
unit  =    3 (00000011)
```

Dal punto di vista pratico esistono numerose possibilità per operare questa "estrazione" del numero in base da quello in una altra base. Il C tende ad effettuare queste operazioni r**endendo come risultato una stringa**, attraverso funzioni come **`printf`**, **`sprintf`**, **`itoa`**, ecc., che sono molto pratiche, ma che hanno la necessità di **una quantità di risorse** che il microcontroller a 8 bit non può sempre mettere a disposizione. Inoltre, data la massa consistente di codice Assembly che producono, ci si trova ad avere tempi di esecuzione inadeguati per molte applicazioni.

Anche il breve algoritmo indicato sopra ha come risultato l'occupazione di oltre 200 bytes di memoria programma. Siccome la memoria in alcuni chip non è moto ampia, ci si può trovare alla ricerca di qualche implementazione che migliori la situazione:
In questo caso, si può fare qualcosa considerando che

`C = A % B` è equivalente a **`C = A - B * (A / B)`**

Pertanto, questo suggerisce una semplice ottimizzazione dell'algoritmo.

```
cent = adval / 100U;
adval - = cent * 100U;
dec = adval / 10U;
adval - = dec * 10U;
unit = adval;
```

Questo "consuma" il contenuto di **adval** che alla fine è uguale al valore dell'unità, però ha un vantaggio in termini di codice Assembly generato.

Se si sostituisce un singolo operatore (**%**) con due operazioni (**- ***), questo rende un certo vantaggio, dato che l'operatore del modulo è eseguito come tre operatori (- * /). Eliminando tre divisioni si ottiene una minore occupazione di memoria programma (circa 170 bytes invece di 200).