PERIFERICHE INTELLIGENTI
colleghiamo un display LCD

Sulla schedina sperimentale che abbiamo costruito c'è un connettore a 14 poli da flat cable per poter collegare un display lcd.
I pin di controllo del display sono in comune con altre funzioni, se guardate attentamente lo schema vedrete che tranne RB7 tutti gli altri segnali che vanno verso il display, vengono anche collegati ad altre periferiche.
Per evitare conflitti sarà ovviamente necessario NON utilizzare le altre periferiche durante il funzionamento del display. Nella fattispecie non potremo pretendere di poter usare le prime 5 uscite del darlington ULN2004 , per questo NON potremo contemporaneamente usare ad esempio il motore stepper visto nel capitolo precedente.

Vediamo ora di capire come è fatto e come funziona un display lcd intelligente come quello da noi utilizzato.

Di display lcd ne esistono di svariate forme, ma il modello in assoluto più flessibile prevede uno schermo composto da una matrice di punti, in questo modo è possibile visualizzare qualunque carattere o simbolo semplicemente accendendo i punti necessari, esattamente come uno schermo CRT di un computer.
Questo genere di display molto in voga negli ultimi anni contiene al suo interno un vero e proprio microprocessore appositamente programmato, e questo ne rende l'uso assai più semplice di quanto possiate immaginare. Questo speciale microprocessore si chiama HD44780 ed è prodotto dalla giapponese HITACHI, è in un contenitore smd 80 pin e si occupa di tutti i dattagli riguardanti il controllo dei singoli pixel del display. Per meno di 10 Euro potete tranquillamente trovare uno di questi moduli completi che contengono un display lcd, un piccolo circuito stampato ed il chip di controllo HD44780. Applicare corrente e leggere/scrivere sul modulo richiede appena 14 connessioni o anche meno se si utilizza un modo speciale di trasmissione dati. Il controller HD44780 può visualizzare fino ad 80 caratteri.
Imparare a programmare questo genere di display richiede un po di esperimenti e di tempo, ma sarete ampiamente ripagati dalla flessibilità di visualizzazione, nonché dal fatto che attualmente quasi tutti i display presenti sul mercato sono assolutamente compatibili con questo vero e proprio standard di fatto.
Questo ci permetterà anche di riciclare in breve tempo tutte le routine di dialogo scritte e che andremo a vedere.
La standardizzazione è tale che anche la piedinatura dei display è sempre uguale ! Comperare di questi oggetti su qualche bancarella di una fiera non richiede mai il corredo di un datasheet.
La tabella 8-1 riporta le connessioni elettriche, si ripete che è uno standard di fatto, anche costruttori differenti utilizzano questa stessa piedinatura. Se quando acquistate un display questo ha un connettore da 14 o 16 pin quasi sicuramente rispetta questo schema di collegamento !

Note sui moduli

Questi moduli lcd a caratteri sono costruiti da molte industrie diverse, Philips, Optrex, Densitron e molte altre, specialmente made in Taiwan.
Il display di uno di questi moduli contiene una o più righe di caratteri. Ogni carattere consiste di una

matrice di punti che tipicamente è di 8x5 , anche se il controller HD44780 potrebbe controllarne fino a 11x5 per poter visualizzare meglio caratteri con la 'gambina' tipo *g b* e *q*.
Il modulo crea i caratteri accendendo i pixel appropriati all'interno di un carattere, la prima riga di pixel è però riservata alla visualizzazione del cursore (se abilitato) lasciando una matrice di 7x5 a disposizione del carattere.
I display sono disponibili in diversi tagli, il più comune fino a poco tempo fa era una riga x 16 caratteri, ma ultimamente il più gettonato appare essere il 2x16, anche 4 righex20 è diventato piuttosto comune.
Più di 80 caratteri richiedono l'uso di un chip supplementare integrato sul modulo, ma dal nostro punto di vista il pilotaggio rimane assolutamente lo stesso.

Alimentazione e luce di sfondo

L'alimentazione è un semplice +5v.. il modulo contiene al suo interno l'oscillatore per il microprocessore ed il pilotaggio dei segmenti LCD. Il consumo totale del dispositivo raramente supera il paio di milliampere. Un ingresso di contrasto (pin3) permette di aggiustare la visualizzazione alle diverse condizioni di luminosità e soprattutto di temperatura.
Alcuni lcd usano una luce di sfondo per permettere la visione con poca luce. Un modulo può essere *riflessivo* (che NON usa luce di sfondo) *trasmissivo* (che deve avere una luce proveniente dal retro) o 'transflettivo' (che può usare la luce dal retro o no). Con quest'ultimo tipo potete accendere o spegnere la luce retrostante per ottimizzare il consumo di corrente.
La luce di illuminazione infatti consuma molto spesso assai più che il resto del modulo. Fino a poco tempo fa un sistema di illuminazione comune consisteva in una luce detta 'EL', ed altro non era che un piccolo sistema fluorescente da alimentare pertanto con qualcosa tipo 100v RMS a circa 400Hz. Ad un basso consumo aggiungeva però la complicazione dell'inverter necessario ad elevare la tensione normalmente disponibile. Una retroilluminazione tipo EL la riconoscete per le due linguette metalliche flessibili che fuoriescono dal fianco del display.

In anni recenti si sono diffusi maggiormente i sistemi di illuminazione a led, consumano assai più corrente ma sono anche molto più facili da pilotare !
In quest'ultimo caso si aggiungono spesso due pin per portare l'alimentazione ai led, rendendo il connettore a 16 poli anziché 14 come standard.

Dentro al controller

L'HD44780 è un vero e proprio microprocessore specializzato con la sua ROM e RAM. Può interpretare 11 istruzioni visibili nella tabella 8-2 . Le istruzioni effettuano compiti come pulire il display, scrivergli un carattere, selezionare una posizione, accendere o spegnere il cursore o leggere lo stato del display. E' importante che capiate queste istruzioni per poter usare il sistema !

Tabella 8-1

Pin	***Simbolo***	***Input/output***	***Funzione***
1	VSS	Input	Alimentazione +5v
2	VDD	Input	Massa
3	V0	input	Regolazione contrasto
4	RS	input	Register Select (1= dato 0= istruzione)
5	R/W	input	Read/Write (1= scrittura 0=Lettura)
6	E	input	Enable (abilitazione: 1= esecuzione)
7	D0	I/O	Bit 0 dato
8	D1	I/O	Bit 1 dato
9	D2	I/O	Bit 2 dato
10	D3	I/O	Bit 3 dato
11	D4	I/O	Bit 4 dato
12	D5	I/O	Bit 5 dato
13	D6	I/O	Bit 6 dato
14	D7	I/O	Bit 7 dato
16	+backlight		alimentazione led retroilluminazione PIN OPZIONALE
17	-backlight		alimentazione led retroilluminazione PIN OPZIONALE

Tabella 8-2

Istruzione	***RS***	***R/W***	***D7***	***D6***	***D5***	***D4***	***D3***	***D2***	***D1***	***D0***	***Funzione***	***Tempo***
Pulisci display	0	0	0	0	0	0	0	0	0	1	Pulisce il display e posiziona sul primo carattere	1.64msec
Display/cursore home	0	0	0	0	0	0	0	0	1	x	Shift = 0 DDRAM = 0	1.64msec
Entry mode	0	0	0	0	0	0	0	1	I/D	S	I/D: Incrementa (1) Decrementa (0) cursore o shift S: Shift on (1) off (0)	40 microsec
Display ON/OFF	0	0	0	0	0	0	1	D	C	B	D: display ON (1) OFF (0) C: cursore ON (1) OFF (0) B: lampeggiamento cursore ON (1) OFF (0)	40 microsec
Shift cursore/display	0	0	0	0	0	1	S/C	R/L	X	X	S/C: shift display (1), o cursore (0) R/L: shift a destra (1) o sinistra (0)	40 microsec
Settaggio funzioni	0	0	0	0	1	DL	N	0	X	X	DL: interfaccia a 8-bit (1) 4-bit (0) N: una riga(0) due righe (1)	40 microsec
settaggio CGRAM	0	0	0	1	CG5	CG4	CG3	CG2	CG1	CG0	indirizzo del generatore di caratteri personalizzato, i dati seguenti lavoreranno su CGRAM	40 microsec
settaggio DDRAM	0	0	1	DD6	DD5	DD4	DD3	DD2	DD1	DD0	indirizzo del generatore di caratteri , i dati seguenti lavoreranno su DDRAM	40 microsec
lettura flag occupato / posizione display	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	lettura flag di display occupato e posizione caratteri	0
scrittura CG/DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	scrive dati (D0/D7) in memorie caratteri	40 microsec
lettura CG/DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	legge dati dalle rispettive memorie	

Aree di memoria

La memoria on chip dell'HD4470 contiene: un generatore di caratteri, ROM, RAM di generatore caratteri personalizzati, DDRAM (display data) un registro istruzioni ed un registro dati.

La **CGROM** contiene i pattern per generare 192 caratteri standard, inclusi l'alfabeto romano in caratteri maiuscoli e minuscoli, numeri, interpunzioni e alcuni simboli matematici, addirittura alcuni caratteri del set giapponese KANA. Questa è ROM e pertanto non può essere alterata.
Ogni carattere in questa memoria così come nella CGRAM ha un indirizzo ad 8 bit.
Per convenienza quelli più comuni seguono lo standard ascii cioè ad esempio il numero 0 è all'indirizzo 30h, la lettera A all'indirizzo 41h e così via.

La **CGRAM** contiene i pattern per fino a 16 caratteri definiti dall'utente (noi !!!) come loghi simboli speciali o altro che potremo creare a partire dalla matrice di 8x5 punti. Per creare un carattere personalizzato basta scrivere una serie di 8 dati ad otto bit nella CGRAM, ogni byte contiene i dati di una riga, e dato che la riga è di 5 pixel, solo i 5 bit meno significativi verranno considerati.
Questa memoria viene persa alla mancanza di corrente e quindi va ricaricata ad ogni avvio del sistema.

C'è poi un registro istruzioni che serve per memorizzare le istruzioni inviate al controller così come un registro dati ne memorizza i dati inviati. Occorrerà selezionare il registro giusto tramite l'apposito pin prima di scrivere qualunque cosa !

La **DDRAM** memorizza fino a 80 codici di 8 bit. Ogni posizione fra le 80 disponibili rappresenta un carattere sul display mentre il codice da 8 bit memorizzatovi rappresenta uno fra i possibili caratteri da visualizzare preso dalla CGROM o CGRAM. Un valore fra 0 e 0Fh pesca il carattere dalla CGRAM mentre valori superiori rappresentano caratteri della CGROM.

All'accensione, in un modello da 2 righe, la posizione in alto a sinistra sul display corrisponde all'indirizzo zero con le seguenti posizioni a destra in ordine crescente. La seconda riga del display comincia all'indirizzo 40h anche se la linea superiore (la prima) ha meno di 40h caratteri.

Le istruzioni permettono di configurare il modulo in modo che all'inserzione di un nuovo carattere la posizione del cursore si incrementi automaticamente di una unità a destra. Basterà quindi scrivere caratteri in successione perché questi vengano visualizzati in ordine da sinistra verso destra.

Al completamento della prima riga però il display NON andrà automaticamente a capo !!!
I dati verranno scritti in ram nelle locazioni successive, che però non corrispondono a nessun pixel visualizzato fisicamente ! La riga successiva inizia solo all'indirizzo 40h ed è a questo indirizzo che occorrerà puntare per poter continuare correttamente la visualizzazione.
In un modello da 2x16 caratteri ad esempio la prima riga inizia a 0 e finisce a 0Fh mentre la seconda comincia all'indirizzo 40h e termina a 4fh.
In alcuni modelli di display monoriga poi (per risparmiare) l'unica riga presente è suddivisa in due righe logiche per cui i primi 8 caratteri sono mappati all'indirizzo 0-7 mentre i restanti otto caratteri sono mappati da 40h a 47h. Con questo tipo di display ci occorrerà usare una istruzione di settaggio DDRAM a 40h prima di scrivere la seconda metà della stessa riga !

In piccoli display dove non tutte le 80h locazioni di DDRAM siano utilizzate, nulla ci vieta di usarle come RAM generica per i nostri scopi ! Sfortunatamente per risparmiare pin spesso NON si collega il segnale di R/W per cui questa funzione non è utilizzabile dato che non possiamo effettuare alcuna lettura dal display.

Leggere e scrivere sul display

Scrivere qualcosa sul display significa seguire questi passi:
- alzare RS a 1 per scrivere dati o abbassarlo per scrivere istruzioni
- abbassare R/W
- porre il dato sul bus D0/D7
- aspettare almeno 140 nanosecondi
- alzare E per almeno 450 nanosecondi
- abbassare E

Le operazioni di lettura sono simili alle scritture eccetto per il fatto che alzeremo il pin R/W a +5v anziché tenerlo basso. Il dato richiesto comparirà sul bus in 320 nanosecondi dall'asserzione di E .

L'HD44780 NON può accettare nuove istruzioni fintanto che non abbia terminato l'esecuzione dell'istruzione precedente, vedere la tabella 8-2 per i tempi di esecuzione. D'altra parte la lettura della BUSY FLAG è sempre possibile e questa sarà settata ad uno quando non è possibile accedere al display mentre tornerà zero quando il display si è liberato dal compito precedente.

Il controllo completo del modulo richiede 8 linee di dato più altre 3 per il controllo. Per risparmiare pin si può configurare il modulo per l'interfacciamento a 4 bit .
Inoltre possiamo anche evitare di leggere la BUSY FLAG semplicemente attendendo un tempo leggermente superiore al richiesto prima di inviare il comando successivo. In questo modo anche il pin R/W può essere lasciato inutilizzato e collegato stabilmente a massa dato che la maggior parte delle operazioni di lettura riguarda la busy flag.

Inizializzazione del modulo

All'accensione il modulo deve essere inizializzato correttamente.
Se l'alimentazione è buona, con la tensione che sale da 0,2 a 4,5 v in 10 millisecondi o meno, il modulo si inizializzerà automaticamente. Se invece l'alimentazione non rispetta queste caratteristiche può essere che il modulo non si avvii correttamente rendendo necessaria l'inizializzazione da parte del nostro software. Per sicurezza è bene quindi effettuare sempre questa procedura, altrimenti il modulo potrebbe non rispondere affatto ad alcun comando !
L'inizializzazione consiste nell'invio consecutivo di tre comandi che settino l'interfaccia a 8 bit, anche se poi noi la useremo in realtà a 4. A tal proposito avete notato che il codice per inizializzare l'interfaccia a 4 o 8 bit dipende dal bit 4 del comando? Furbi vero ? Nell'interfaccia a 4 bit le 4 linee usate sono dalla DB4 alla DB7 ;) Se le 4 restanti linee verranno collegate a massa come indicato dal costruttore saremo a posto perché riusciremo comunque a lavorare sul bit necessario a settare il modo 4 bit !

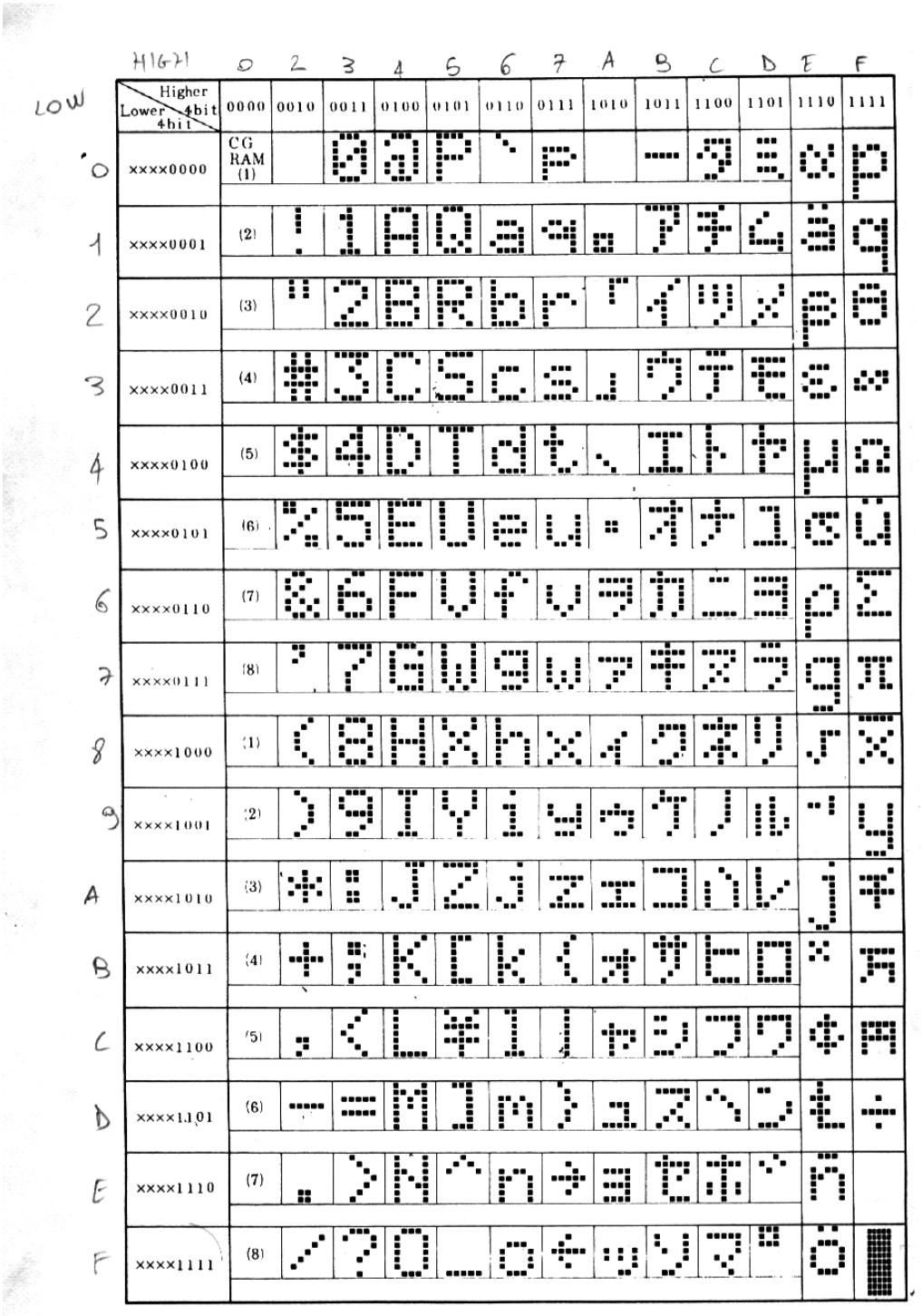
Il nostro programma comincerà quindi inviando tre volte il codice 30h a distanza di almeno 40

microsecondi... meglio 50...

Dopo questo potremo inizializzare il modulo con l'interfaccia a 4 bit, selezionare il tipo di cursore... accendere il display... si ! di default il sistema si avvia con il display spento !
poi selezioneremo il modo a 1 o 2 righe etc...

Fra i vari comandi di inizializzazione, probabilmente l'ultimo è quello per posizionare il cursore nella posizione di HOME cioè l'indirizzo 0 ovvero primo carattere in alto a sinistra.
Da quel momento, scrivendo un dato nella memoria DDRAM il carattere corrispondente comparirà sul display, scrivendone un secondo comparirà immediatamente a destra e così via fino al completamento della riga.

La corrispondenza fra caratteri e codici è data nella seguente tabella:



Se conoscete la codifica ASCII ne scoprirete la somiglianza, almeno per numeri, lettere e simboli di uso comune. Tenete ben presente questa tabella... ci servirà anche per la comunicazione seriale RS232.

Torniamo sull'uso dell'interfaccia a 4 bit poiché la cosa è per noi importante.
Il costruttore dice di usare le linee DB4/DB7 collegando a massa le inutilizzate DB0/DB3.
I dati vengono trasferiti con due nibble da 4 bit facendo cioè due operazioni di scrittura/lettura consecutive in modo da muovere gli 8 bit richiesti da un comando o dato.
Il primo nibble da trasmettere è la parte alta del dato seguita dal nibble basso.
Per trasferire ad esempio il dato 5Ah occorrerà:

- alzare RS a 1 per scrivere dati o abbassarlo per scrivere istruzioni
- abbassare R/W
- porre il nibble alto cioè 5 sul bus dati D4/D7
- aspettare almeno 140 nanosecondi
- alzare E per almeno 450 nanosecondi
- abbassare E
- porre il nibble basso cioè 0Ah sul bus dati D4/D7
- alzare E per almeno 450 nanosecondi
- abbassare E

Come si vede è un pochino più lungo ma, una volta scritto in una subroutine, la sua lunghezza non ci preoccuperà più permettendoci nel contempo di recuperare 4 pin preziosi del pic.

Routine implementate

Per poter gestire comodamente il nostro display ci converrà creare tre routine che richiameremo durante l'uso nel resto del programma.

La prima DispInit serve per inizializzare il modulo. E' da richiamare prima di ogni altra operazione sul display.

DispCmd permette di scrivere un comando sul display che gli sia passato in W

DispData scrive un dato (carattere solitamente) passato in W

Segue ora un esempio di programma piuttosto lungo e completo, che incorpora queste tre routine.

Display_LCD

La lunghezza del programma comincia ad essere considerevole, quindi caricare l'esempio già fatto dal cd mi sembra doveroso...

```
list P=16f84A , R=DEC
#INCLUDE P16F84A.inc

__FUSES  _CP_OFF&_WDT_OFF&_XT_OSC

CBLOCK 0ch
Timer1          ;Subcontatore per DelayP
Timer2          ;Contatore per DelayP
Counter         ;Contatore valore da mostrare a display
CounterOld      ;valore di Contatore precedente a UpdateDisplay
DispTemp        ;contenitore temporaneo per calcoli routine display
DispCnt         ;contatore generico per gestione display
X2asctmp        ;valore temporaneo per conversione hex2asc
```

```
        TabTemp         ;memoria temporanea per implementazione tabelle costanti
        ENDC

CounterPreset=1         ;valore presettato di contatore
AntiDebounce=200        ;timer antirimbalzo per pulsanti

;inizio programma dopo un reset

        org 0

;inizializzazione sistema e variabili

        bsf STATUS,RP0
        movlw 01110001b         ;0=uscita 1=ingresso
        movwf TRISB
        movlw 11100000b
        movwf TRISA
        bcf STATUS,RP0

        movlw CounterPreset     ;setta il contatore al valore di default
        movwf Counter
        movwf CounterOld
        incf CounterOld,f       ;trucco per aggiornare la visualizzazione da subito ... ;)

;inizializzazione display

        call DispInit

;Frase iniziale sul display "Contatore: "

        clrf DispCnt            ;azzera contatore carattere messaggio
Msg1
        movf DispCnt,w          ;preleva puntatore carattere messaggio
        call TabellaMsg1        ;preleva carattere relativo
        call DispD              ;scrivilo sul display
        incf DispCnt,f          ;incrementa puntatore ai caratteri
        movlw 11                ;10 caratteri da visualizzare (1 in +... prima incrementiamo POI
controlliamo)
        subwf DispCnt,w         ;sottrai contatore caratteri messaggio...
        btfss STATUS,Z          ;...per vedere se siamo alla fine
        goto Msg1


;loop principale programma

Loop
```

```
        call Button             ;aggiorna il valore del contatore in base allo stato dei pulsanti

        call DispUpdate

        movlw AntiDebounce      ;antirimbalzo pulsanti di 20 millisecondi
        call DelayP

        goto Loop               ;ricomincia da capo


;***********************************************
;subroutine Button , legge lo stato dei pulsanti e modifica il valore del contatore
;***********************************************

Button

        bsf PORTB,1             ;accende led di debug

        ;pulsante 1: incrementa contatore
        btfsc PORTB,4           ;leggi lo stato del pulsante 1
        incf Counter,f          ;se premuto incrementa il valore del settaggio tempo

        ;pulsante 2: decrementa contatore
        btfsc PORTB,5           ;controlla ora il pulsante di decremento
        decf Counter,f          ;decrementa il contatore

        ;pulsante 3: azzera contatore
        movlw CounterPreset             ;carica il valore di default contatore
        btfsc PORTB,6           ;controlla il terzo pulsante
        movwf Counter           ;se premuto carica il valore di preset nel contatore


;attendiamo il rilascio dei pulsanti eventualmente premuti

Butt1

        movlw 01110000b         ;maschera di AND per stato pulsanti
        andwf PORTB,w           ;maschera i bit che non servono e deposita il risultato in W non
                                ;PORTB !!!
        btfss STATUS,Z          ;se risultato = zero, nessun pulsante premuto
        goto Butt1

        bcf PORTB,1             ;spegne led 1 di debug

        return                  ;esce dalla subroutine
```

```
;*******************************************
;subroutine DispUpdate , aggiorna display solo se il valore di Contatore è variato
;*******************************************
DispUpdate

        bsf PORTB,2         ;accende led 2 di debug

        movf Counter,w      ;preleva contatore attuale
        subwf CounterOld,w  ;sottraigli il valore del loop precedente
        btfsc STATUS,Z      ;controlla se =0 cioè se valori ancora uguali
        goto DispUpdateEnd  ;esci se i valori sono uguali

        movlw 8ch           ;comando posizione cursore sul numero da aggiornare
        call DispC          ;invia comando al display

        swapf Counter,w     ;preleva il nibble alto in quello basso di W
        andlw 0fh           ;maschera i bit che non servono
        call Hex2asc        ;converti in ascii visualizzabile
        call DispD          ;mostra a display

        movlw 0fh           ;maschera i bit che non servono
        andwf Counter,w
        call Hex2asc        ;converti in ascii visualizzabile
        call DispD          ;mostra a display

        movf Counter,w      ;aggiorna il valore di CounterOld al nuovo valore
        movwf CounterOld

DispUpdateEnd

        bcf PORTB,2         ;spegne led 2 di debug

        return



;*******************************************
;* DispInit: Inizializza il display all'accensione
;*******************************************
DispInit

;come da datasheet inviamo 3 comandi di settaggio interfaccia ad 8 bit
;prima per sicurezza attendiamo 10 msec che il chip si accenda per benino...

        movlw 100           ;10 msec di ritardo
        call DelayP
```

```
        movlw 00000011b     ;comando display per interfaccia ad 8 bit, lavoriamo a 4 bit !!!
        movwf PORTA

        bsf PORTB,7         ;abilita display
        nop                 ;meglio mettere una pausa di 1 microsecondo
        bcf PORTB,7         ;disabilita display

        movlw 1             ;attendi almeno 40 microsecondi, 100 va meglio !
        call DelayP

        bsf PORTB,7         ;idem come sopra
        nop
        bcf PORTB,7

        movlw 1
        call DelayP

        bsf PORTB,7
        nop
        bcf PORTB,7

        movlw 1
        call DelayP

;Ora il display è ben inizializzato, commutiamo l'interfaccia a 4 bit

        movlw 00000010b     ;comando display per interfaccia a 4 bit
        movwf PORTA
        bsf PORTB,7         ;abilita display
        nop                 ;meglio mettere una pausa di 1 microsecondo
        bcf PORTB,7         ;disabilita display
        movlw 1             ;attendi almeno 40 microsecondi, 100 va meglio !
        call DelayP

;da questo momento lavoriamo a 4 bit, finalmente ! ora possiamo usare DispC per inviare
qualunque comando

        movlw 5             ;5 comandi da inviare al display
        movwf DispCnt
DispIniLoop
        movf DispCnt,w      ;preleva numero comando da inviare
        call Tabella1       ;preleva il relativo comando dalla tabella
        call DispC          ;invia comando al display
        movlw 20            ;attendi circa due millisecondi pari alla durata del comando + lungo
        call DelayP
        decfsz DispCnt,f    ;decrementa contatore numero comandi
```

```
        goto DispIniLoop    ;reitera se ci sono ancora comandi da inviare

        return



;*********************************************
;DispC: invia un comando al display LCD
;*********************************************
DispC
        movwf DispTemp      ;ripone al sicuro il comando
        swapf DispTemp,w    ;inverte i nibble e ne recupera il contenuto in W
        movwf PORTA         ;invia il comando al display
        bcf PORTA,4         ;azzera bit 4 per selezione RS=0 -> istruzione
        bsf PORTB,7         ;abilita il display
        nop
        bcf PORTB,7         ;disabilita il display
        movf DispTemp,w     ;preleva comando
        movwf PORTA         ;invia il comando al display
        bcf PORTA,4         ;azzera bit 4 per selezione RS=0 -> istruzione
        bsf PORTB,7         ;abilita il display
        nop
        bcf PORTB,7         ;disabilita il display

        movlw 1             ;attende 100 microsecondi per dare il tempo al display di
                            ;digerire il comando
        call DelayP

        return



;*********************************************
;DispD: invia un dato al display LCD
;*********************************************
DispD
        movwf DispTemp      ;ripone al sicuro il comando
        swapf DispTemp,w    ;inverte i nibble e ne recupera il contenuto in W
        movwf PORTA         ;invia il comando al display
        bsf PORTA,4         ;setta bit 4 per selezione RS=1 -> dato
        bsf PORTB,7         ;abilita il display
        nop
        bcf PORTB,7         ;disabilita il display
        movf DispTemp,w     ;preleva comando
        movwf PORTA         ;invia il comando al display
        bsf PORTA,4         ;setta bit 4 per selezione RS=1 -> dato
        bsf PORTB,7         ;abilita il display
        nop
```

```
        bcf PORTB,7         ;disabilita il display

        movlw 1             ;attende 100 microsecondi per dare il tempo al display di
                            ;digerire il dato
        call DelayP

        return


;*********************************************
;subroutine DelayP: Ritarda l'esecuzione di 100 microsecondi per ogni unità di W
;+ 4 microsecondi aggiuntivi sempre presenti
;*********************************************

DelayP
        movwf Timer2        ;deponi il valore passato in W nel contatore che useremo
        movlw 19
        movwf Timer1        ;presetta subtimer1 19 ciclix 5 istruzioni = 95 microsecondi
                            ;cui vanno aggiunti i cicli esterni al loop D1

D1      nop
        nop
        decfsz Timer1,f
        goto D1
        movlw 19            ;occorre ogni ciclo ricaricare Timer 1 con il preset
        movwf Timer1
        nop
        decfsz Timer2,f
        goto D1
        return



;*********************************************
;hex2asc: codifica un nibble in w in un byte ascii sempre in w
;*********************************************

Hex2asc
        movwf X2asctmp
        sublw  9            ;guarda se > 9
        btfss   STATUS,C
        goto    Hex2asc1    ;oltre se lo e'
        movf   X2asctmp,w   ;recupera il dato
        addlw  30h          ;somma 30 per trasformarlo in ascii
        return

Hex2asc1
```

```
        movf   X2asctmp,w   ;recupera il dato
        addlw  37h          ;somma 37h per avere da A ascii in poi (41h...)
        return


;*********************************************
;Tabelle varie
;*********************************************

        org 300h            ;ultimo blocco da 256 byte prima di 3FFh che è il fine ROM

;tabella valori per inizializzazione display

Tabella1
        movwf TabTemp           ;riponi W al sicuro
        movlw high(Tabella1)    ;parte alta dell'indirizzo programma corrente
        movwf PCLATH            ;deponila in PCLATH
        movf TabTemp,w          ;recupera puntatore alla tabella salvato precedentemente
        addwf PCL,f             ;W deve contenere il puntatore alla tabella
        nop                             ;il numero zero NON lo usiamo
        retlw 00000001b         ;pulisce il display e riporta il cursore a posizione zero
        retlw 00000110b         ;incrementiamo posizione caratteri dopo ogni trasferimento,
                                ;non shiftiamo il display
        retlw 00001100b         ;display acceso, cursore spento e non lampeggiante
        retlw 00010100b         ;shifta il cursore verso destra
        retlw 00101000b         ;display a due righe, nota che dobbiamo riconfermare il modo
4 bit !

TabellaMsg1
        movwf TabTemp
        movlw high(TabellaMsg1)
        movwf PCLATH
        movf TabTemp,w
        addwf PCL,f
        retlw 'C'
        retlw 'o'
        retlw 'n'
        retlw 't'
        retlw 'a'
        retlw 't'
        retlw 'o'
        retlw 'r'
        retlw 'e'
        retlw ':'
        retlw ' '

        end
```

Analizziamo questo lungo programma un pezzo alla volta !

DispInit:

Cominciamo con la subroutine *DispInit* .
Questa routine la potete vantaggiosamente copiare per riutilizzarla tale e quale in futuro nei vostri programmi.

Per prima cosa attendiamo qualcosa tipo 10 millisecondi per dare al display il tempo di inizializzarsi correttamente dal momento in cui l'alimentazione viene fornita. Dal data sheet si vede che il comando di inizializzazione completa richiede circa 1,64 millisecondi. Per sicurezza attendiamo alcune volte tanto questo valore .
Sempre dal datasheet avevamo visto che occorreva per prima cosa inviare tre comandi di inizializzazione interfaccia a 8 bit.
Noi usiamo l'interfaccia a 4 bit, ma il datasheet è abbastanza perentorio su questo, i tre comandi vanno comunque inviati.
Nello schema della nostra scheda di sperimentazione si vede che abbiamo collegato stabilmente a massa il quattro bit bassi del display. Il comando che controlla il tipo di interfacciamento a 4 o 8 bit impiega solo il nibble alto ! quindi è comunque perfettamente possibile inviare correttamente il comando con una sola scrittura. Se inviamo su PORTA il nibble alto del comando che è:

00000011b

in realtà al display arriva

00110000b

dato che il nibble basso del display è collegato appunto a massa mentre il nibble alto è cablato sul nibble basso di PORTA.

Dopo ogni invio di un comando occorrerebbe leggere il bit *busy flag* con un comando di lettura stato. Ma noi NON abbiamo volutamente collegato la linea R/W, necessaria a leggere valori dal display verso il pic, per risparmiare pin. Come facciamo quindi a vedere se il display si è liberato ? Semplice ! consultiamo il datasheet e scopriamo che ad ogni comando corrisponde sempre un tempo di esecuzione. Basterà attendere almeno quel tempo, se non di più, per avere la certezza che di quel comando il display ne ha terminato l'esecuzione.

Abbiamo quindi creato la ruotine *DelayP* appositamente per questo scopo, dove la *P* del nome stà per *programmabile*. Il ritardo di tempo dipende dal valore che assume il registro W al momento della chiamata.
La routine è strutturata in modo da 'perdere' 100 microsecondi di tempo per unità di W passato. Significa che se carichiamo 1 in W prima di effettuare la chiamata allora perderemo 100 microsecondi, mentre ne perderemo 1500 se avremo caricato in W il valore 15.
Il ritardo più piccolo che incontriamo nel datasheet del display è di 40 microsecondi. Dato che non abbiamo necessità di grandi performances in termini di tempo e per semplificarci i calcoli stabiliamo di perdere 100 microsecondi come tempo di base.
Inviamo il comando di inizializzazione ad 8 bit con la sequenza che segue:

```
1     movlw 00000011b  ;comando display per interfaccia ad 8 bit,
                       ;lavoriamo a 4 bit !!!
2     movwf PORTA

3     bsf PORTB,7      ;abilita display
4     nop              ;meglio mettere una pausa di 1 microsec.
5     bcf PORTB,7      ;disabilita display

6     movlw 1          ;attendi almeno 40 microsecondi, 100 va meglio!
7     call DelayP
```

Alla riga 1 carichiamo il nibble alto del comando di inizializzazione ad 8 bit del display.
Successivamente lo trasferiamo su PORTA cioè fisicamente sulla porta di comunicazione verso il display.
Alla riga 3 alziamo il bit di Enable del modulo lcd, attendiamo un microsecondo alla riga 4 per dare con sicurezza il tempo alle linee di assestarsi e poi riabbassiamo il segnale Enable alla riga 5. Sul fronte di discesa di questo Enable il modulo effettuerà la lettura del comando.
Come preannunciato occorre adesso attendere almeno 40 microsecondi. E' quanto facciamo alle righe 6 e 7 caricando il valore minimo in W e chiamando *DelayP* che perderà quindi 100 microsecondi.

Queste semplici operazioni vanno ripetute per tre volte, come potete vedere nel listato principale. Lo facciamo semplicemente ripetendo il pezzo di codice qui sopra riportato. Per solo tre scritture necessarie, non conviene in questo caso impostare un ciclo di ripetizione con un contatore... basta usare il copia/incolla sul vostro pc per ottenere velocemente quanto serve !

Proseguendo nell'inizializzazione del display occorre ora comunicargli che dovremo in realtà lavorare a 4 bit ! Il comando di interfaccia a 8 bit è stato comunque possibile inviarlo, ma SOLO perché i bit significativi del comando risiedevano nel nibble alto. Se avessimo voluto anche settare nel modulo la visualizzazione a due righe non avremmo potuto dato che il bit 3 del bus dati è stabilmente collegato a massa !
Settando il modo di funzionamento a 4 bit allora potremo, dividendo la scrittura dei dati in due volte consecutive, inviare dati completi ad 8 bit.
E' quanto facciamo nel seguente pezzo di codice:

```
;Ora il display è ben inizializzato, commutiamo l'interfaccia a 4 bit

1     movlw 00000010b  ;comando display per interfaccia a 4 bit
2     movwf PORTA
3     bsf PORTB,7      ;abilita display
4     nop              ;meglio mettere una pausa di 1 microsec.
5     bcf PORTB,7      ;disabilita display
6     movlw 1          ;attendi almeno 40 microsecondi, 100 va meglio !
7     call DelayP
```

Alle righe 1 e 2 inviamo il nibble alto del comando di settaggio a 4 bit.
Abilitiamo poi il display per farglielo leggere alle righe 3,4 e 5 .
Dopo l'attesa delle righe 6 e 7 possiamo con sicurezza asserire che il display lavora a 4 bit. I comandi, che restano gli stessi, vanno inviati mediante due scritture successive, prima il nibble alto e successivamente quello basso.
Per semplificarci la vita creiamo una routine chiamata *DispC* ovvero *DISPlayComando* . Serve per

inviare al display un comando passato in W prendendosi cura di splittare il comando passato in due scritture consecutive con tutte le temporizzazioni necessarie.
La routine segue:

```
DispC
1     movwf DispTemp  ;ripone al sicuro il comando
2     swapf DispTemp,w      ;inverte i nibble e ne recupera il contenuto
                            ;in W
3     movwf PORTA     ;invia il comando al display
4     bcf PORTA,4     ;azzera bit 4 per selezione RS=0 -> istruzione
5     bsf PORTB,7     ;abilita il display
6     nop
7     bcf PORTB,7     ;disabilita il display

8     movf DispTemp,w ;preleva comando
9     movwf PORTA     ;invia il comando al display
10    bcf PORTA,4     ;azzera bit 4 per selezione RS=0 -> istruzione
11    bsf PORTB,7     ;abilita il display
12    nop
13    bcf PORTB,7     ;disabilita il display

14    movlw 1         ;attende 100 microsecondi per dare il tempo al
                      ;display di digerire il comando
15    call DelayP

16    return
```

Per prima cosa appoggiamo il comando passato in una locazione di memoria RAM dove resterà al sicuro durante le successive operazioni che coinvolgeranno anche W. Questa locazione inoltre ci occorre anche perché la successiva istruzione che andiamo a vedere lavora unicamente su una locazione RAM e non su W.

Alla riga 2 incontriamo quindi un comando nuovo: *swapf* .
Questa istruzione scambia i due nibble di un registro RAM. Dopo l'esecuzione il nibble alto apparirà al posto di quello basso e viceversa. Ci occorre effettuare questa operazione poiché dobbiamo inviare all'LCD per prima cosa il nibble alto del comando ma la porta su cui lavoriamo è nel nible basso di PORTA. Scambiando i due nibble e prelevando il risultato in W (flag *,w* dopo il comando) avremo quanto ci serve. Notare che il contenuto di DispTemp resterà inalterato dopo l'operazione.
Alla riga 3 inviamo quanto appena detto sulla porta fisica di comunicazione con il display.
Essendo questo un comando, è importante settare correttamente il bit di selezione comandi sull'LCD. E' quanto facciamo alla riga 4 in cui azzeriamo lo stato del pin che andrà al display. Questa è una operazione importante dato che abbiamo appena scritto in blocco l'intera PORTA con un valore di 8 bit in cui il bit 4 potrebbe essere 1 o 0 a seconda del dato prelevato.

Scriviamo il comando sul display alle righe 5,6 e 7 come già visto in precedenza.

Trasferito il nibble alto è ora la volta di quello basso.
Per questo NON ci occorre scambiare alcun bit del comando da inviare. Come già spiegato la porta di comunicazione verso il display risiede nel nibble basso di PORTA, scrivendovi sopra il comando così come era stato passato alla routine è più che sufficiente.

E' invece importante resettare a zero il bit 4 di PORTA per azzerare la linea di selezione Comando/dato verso il modulo.

Le righe 11,12 e 13 scrivono il dato sul modulo esattamente come alle righe 5,6 e 7.

Resta solo da attendere 100 microsecondi mediante *DelayP* per completare l'operazione e tornare al programma chiamante.

Esattamente identica è la successiva routine *DispD* (DISPlayDato) che segue. L'unica differenza è che verrà impiegata per inviare dati (cioè caratteri da visualizzare) anziché comandi al modulo e quindi setta ad uno la linea R/S del display con :

```
     bsf PORTA,4         ;setta bit 4 per selezione RS=1 -> dato
```

prima di scrivere ogni nibble.

Avendo creato queste due routine possiamo sollevarci di ogni problema riguardante il dialogo con il modulo, semplicemente carichiamo in W il valore da inviare e chiamiamo una delle due routine a seconda del compito da eseguire.

Torniamo all'inizializzazione del display.
Dopo aver settato ilmodo di funzionamento a 4 bit possiamo da questo momento in poi usare le due routine DispC e DispD per inviare i dati.
E usiamo subito DispC per completare il settaggio del display prima di poterlo definitivamente utilizzare.
IL codice segue:

```
1     movlw 5          ;5 comandi da inviare al display
2     movwf DispCnt
DispIniLoop
3     movf DispCnt,w   ;preleva numero comando da inviare
4     call Tabella1    ;preleva il relativo comando dalla tabella
5     call DispC       ;invia comando al display
6     movlw 20         ;attendi circa due millisecondi pari alla durata
                       ;del comando + lungo
7     call DelayP
8     decfsz DispCnt,f ;decrementa contatore numero comandi
9     goto DispIniLoop ;reitera se ci sono ancora comandi da inviare
10    return
```

Inviamo cinque comandi al modulo, per farlo usiamo un loop con il contatore *DispCnt*.
Lo inizializziamo a 5 alle righe 1 e 2 poi cominciamo il loop vero e proprio.
Alla riga 3 preleviamo il valore attuale del contatore in W per poi chiamare *Tabella1* a fare la conversione fra numero del contatore e comando da inviare. in *Tabella1* abbiamo memorizzato in ordine tutti i 5 comandi da inviare al display. Occorre solo badare al fatto che *DispCnt* conterà da 5 a 1 cioè in ordine inverso ! I comandi vanno quindi messi in tabella al contrario, partendo dall'ultimo.
Dopo aver prelevato il comando dalla tabella1 lo inviamo al display con la chiamata a *DispC* di riga 5 , dopotutto *Call Tabella1* lascia il valore estratto in W mentre *DispC* si aspetta il comando per il

display sempre in W che viene così ad essere utilizzato come registro di scambio.
In condizioni normali *DispC* attende 100 microsecondi l'esecuzione di ogni comando, il che per la maggior parte di essi è ampiamente sufficiente. Ma qui, fra gli altri, eseguiamo un comando particolare. Il comando `00000001b` che vedremo in tabella1 ha bisogno di 1,6 millisecondi per essere eseguito !
Per non complicare il programma decidiamo di perdere comunque 2 millisecondi dopo ogni comando inviato. Può sembrare uno spreco... in totale però perdiamo circa 10 millisecondi... e tutto questo una sola volta all'accensione del sistema !

Alla riga 8 decrementiamo il contatore dei comandi ed eseguiamo la successiva riga 9 solo se DispCnt non si è azzerato e cioè se ci sono ulteriori comandi da processare.

Al termine di questo loop avremo definitivamente inizializzato il display a nostro piacimento. Se in un altro progetto vorremo, ad esempio, il cursore acceso potremo inviare successivamente il relativo comando oppure modificare la Tabella 1.
Questo è il settaggio standard che chi scrive ha sempre utilizzato.
All'esecuzione della successiva istruzione *Return* avremo il display acceso ed il cursore di scrittura dati posizionato in alto a sinistra. Inviando dati (caratteri ascii) al modulo questi compariranno in ordine da sinistra verso destra sulla prima riga.

Vediamo ora i comandi inviati nel dettaglio.
Per alcuni di questi non esiste un'ordine preferenziale, ma per altri sì ... andiamo ad analizzarli vedendo la *Tabella1*:

```
Tabella1
      movwf TabTemp          ;riponi W al sicuro
      movlw high(Tabella1)   ;parte alta dell'indirizzo programma corrente
      movwf PCLATH           ;deponila in PCLATH
      movf TabTemp,w         ;recupera puntatore alla tabella salvato
                             ;precedentemente
      addwf PCL,f            ;W deve contenere il puntatore alla tabella
      nop                    ;il numero zero NON lo usiamo
1     retlw 00000001b        ;pulisce il display e riporta il cursore a
                             ;posizione zero
2     retlw 00000110b        ;incrementiamo posizione caratteri dopo ogni
                             ;trasferimento,
                             ;non shiftiamo il display
3     retlw 00001100b        ;display acceso, cursore spento e non
                             ;lampeggiante
4     retlw 00010100b        ;shifta il cursore verso destra
5     retlw 00101000b        ;display a due righe, nota che dobbiamo
                             ;riconfermare il modo
```

Ignoriamo per ora la parte iniziale del codice di Tabella1, verrà illustrato più avanti.
Concentriamoci invece sui comandi da inviare al display.
I commenti presenti a destra sono autoesplicativi, si consiglia caldamente di metterli sempre !
Per primo verrà quindi inviato il comando della riga 5. Questo dice al processore del display che ha due righe di pixel da comandare. Dobbiamo pensare che il processore HD44780 è universale per tutte le versioni di display ! Non puo sapere a priori che tipo di hardware vi è collegato, glielo diciamo noi con questo comando, notate che nello stesso comando è presente il tipo di interfaccia a 8 o 4 bit settato prima... ovviamente deve essere uguale !

Riga 4: il bit 3 (0) indica che all'introduzione di un nuovo carattere dobbiamo spostare il cursore e non shiftare il contenuto dell'intero display, mentre al bit 2 (1) indichiamo che lo spostamento appena indicato deve essere verso destra.

Riga 3: Il bit 2 (1) indica che il display deve essere acceso, spento non ci occorrerebbe gran che ! Il bit 1 (0) tiene spento il cursore mentre il bit 0 (0) specifica che se il cursore fosse acceso NON sarebbe lampeggiante.

Riga 2: specifica il modo di introduzione nuovi caratteri. Ad ogni nuovo dato scritto il cursore viene spostato verso destra e i dati eventualmente presenti NON vengono spostati ma sovrascritti.

Riga 1: L'intero display viene cancellato ed il cursore viene riportato sul primo carattere in alto a sinistra. Questo comando richiede molto tempo per essere eseguito...

Dopo l'inizializzazione

```
;Frase iniziale sul display "Contatore: "

1     clrf DispCnt          ;azzera contatore carattere messaggio
Msg1
2     movf DispCnt,w   ;preleva puntatore carattere messaggio
3     call TabellaMsg1 ;preleva carattere relativo
4     call DispD       ;scrivilo sul display
5     incf DispCnt,f   ;incrementa puntatore ai caratteri
6     movlw 11         ;10 caratteri da visualizzare (1 in +... prima
                       ;incrementiamo POI controlliamo)
7     subwf DispCnt,w  ;sottrai contatore caratteri messaggio...
8     btfss STATUS,Z   ;...per vedere se siamo alla fine
9     goto Msg1
```

Tornando all'inizio del programma, dopo aver inizializzato il display con l'apposita routine, cominciamo subito a scrivere qualcosa sul nostro modulo LCD !
Inizializziamo un contatore che, puntando ad una tabella, scriva successivamente i caratteri in essa presenti sul display mostrando quindi un messaggio codificato in ascii.
Di loop di conteggio ne abbiamo già visti diversi e questo non fa certo eccezione.
Usiamo *DispCnt* come contatore (già usato anche in DispInit) per puntare al contenuto di *TabellaMsg1* e prelevarne il contenuto da inviare al display come dato. Nella tabella codifichiamo i dati direttamente in ascii mettendoli fra virgolette:

```
      retlw 'C'
      retlw 'o'
      retlw 'n'
      retlw 't'
      retlw 'a'
      retlw 't'
      retlw 'o'
      retlw 'r'
      retlw 'e'
      retlw ':'
      retlw ' '
```

Il segreto di questo sta nella codifica ASCII che è universale. Il vostro PC ed il display la utilizzano entrambi e quindi un carattere visualizzato correttamente sul vostro pc altrettanto lo sarà nel display.
Le virgolette fra cui i caratteri sono racchiusi sono una convenzione dell'assemblatore per indicare di codificare il dato appunto in questo formato.
La codifica ASCII prevede ad esempio che il carattere *0* sia rappresentato dal numero 48 (30h).
Se scrivo in assembler *'0'* l'assemblatore lo sostituisce appunto con il numero 48.
Dato che anche il display segue la stessa codifica possiamo facilmente visualizzare caratteri standard senza doverli faticosamente convertire manualmente.

In questa serie di dati per il display codifichiamo la frase '*Contatore: '* spazio finale compreso !

Dopo aver prelevato un carattere dalla tabella lo inviamo al display alla riga 4 del listato precedente e poi incrementiamo il puntatore al carattere successivo.
Alla riga 6 controlliamo se abbiamo già inviato tutti gli 11 caratteri al display, il controllo viene effettuato dopo l'incremento della variabile, ricordiamo però che il contatore è stato inizializzato, e quindi vi inizia a contare, a zero, 11 è quindi il numero corretto da sottrarre e non 12 per avere 11 cicli totali.
Una volta che il risultato della sottrazione sia zero (`btfss STATUS,Z`) allora il contatore è giunto alla fine del conteggio necessario e possiamo proseguire.

Al termine avremo la scritta 'Contatore: ' presente sul display a partire da sinistra sulla prima riga.

Il loop principale del programma

```
;loop principale programma

Loop

      call Button           ;aggiorna il valore del contatore in base allo
                            ;stato dei pulsanti

      call DispUpdate

      movlw AntiDebounce    ;antirimbalzo pulsanti di 20 millisecondi
      call DelayP

      goto Loop        ;ricomincia da capo
```

Si occupa di leggere lo stato dei pulsanti. Se qualcuno di questi è stato premuto allora incrementiamo o decrementiamo un valore numerico che poi visualizzeremo a display.
Un ritardi di 20 millisecondi ad ogni ciclo eviterà che il rimbalzo meccanico dei pulsanti faccia incrementare o decrementare eccessivamente il contatore ad ogni pressione.

Avendo modularizzato tutti questi compiti il loop principale appare molto semplice e ci consente di avere meglio sotto controllo cosa fa il nostro programma.

Button

```
1     bsf PORTB,1           ;accende led di debug
```

```
      ;pulsante 1: incrementa contatore
2     btfsc PORTB,4    ;leggi lo stato del pulsante 1
3     incf Counter,f   ;se premuto incrementa il valore del settaggio tempo

      ;pulsante 2: decrementa contatore
4     btfsc PORTB,5    ;controlla ora il pulsante di decremento
5     decf Counter,f   ;decrementa il contatore

      ;pulsante 3: azzera contatore
6     movlw CounterPreset         ;carica il valore di default contatore
7     btfsc PORTB,6    ;controlla il terzo pulsante
8     movwf Counter    ;se premuto carica il valore di preset nel contatore


;attendiamo il rilascio dei pulsanti eventualmente premuti

Butt1

9     movlw 01110000b  ;maschera di AND per stato pulsanti
10    andwf PORTB,w    ;maschera i bit che non servono e deposita il
                       ;risultato in W non PORTB !!!
11    btfss STATUS,Z   ;se risultato = zero, nessun pulsante premuto
12    goto Butt1

13    bcf PORTB,1      ;spegne led 1 di debug
```

Alla riga 1 accendiamo un led per scopo di debug. Potremo vedere facilmente quando il programma è dentro questa routine.

Dalla riga 2 alla 8 incrementiamo, decrementiamo o azzeriamo il *Counter* a seconda del pulsante premuto.
Per evitare che il contatore incrementi o decrementi continuamente se i pulsanti vengo mantenuti premuti abbiamo creato il codice fra le righe 9 e 12.
Adottiamo qui una tecnica nuova detta di **mascheramento a bit**.
In pratica leggiamo l'intera porta contenente i bit dei pulsanti che vogliamo controllare e la mettiamo in AND con una **maschera** che contiene bit a 0 nelle posizioni che non vogliamo controllare.
I pulsanti sono collegati nel nostro circuito ai pin RB4 RB5 ed RB6. Leggendo l'intera PORTB leggiamo lo stato di questi pulsanti, ma anche altri bit che al nostro scopo non interessano.
Se mettiamo in AND il valore appena letto di PORTB con il valore 01110000b il risultato in W sarà un valore che avrà ad 1 solamente i bit eventualmente attivi alle posizioni 4 5 e 6. Gli altri bit 0,1,2,3 e 7 verranno comunque forzati a zero dalla operazione di AND di cui ricordiamo brevemente la tabella della verità:

A	***B***	***output***
0	0	0
1	0	0
0	1	0
1	1	1

Applicando bit per bit questa tabella si vede che dove la maschera 01110000b contiene uno zero, il risultato è sempre e solo zero, mentre se il valore è uno allora l'uscita dipende dallo stato dell'altro termine dell'operazione cioè nel nostro caso PORTB. Riportando la tabella al nostro caso avremo:

bit	*maschera*	*PORTB*	*risultato in W*
7	0	x	sempre zero
6	1	uno o zero	uno o zero = RB6
5	1	uno o zero	uno o zero = RB5
4	1	uno o zero	uno o zero = RB4
3	0	x	sempre zero
2	0	x	sempre zero
1	0	x	sempre zero
0	0	x	sempre zero

Potremo con un solo test controllare se tutti i tre pulsanti sono rilasciati cioè W=0

Alla riga 9 carichiamo in W la maschera di bit necessaria.
Alla riga 10 effettuiamo l'operazione di AND. E' importante in questo caso la flag *,W* !!!
Se non la mettessimo, il risultato dell'operazione verrebbe riposto nuovamente in PORTB andando a mettere degli zeri su tutti i pin in cui la maschera di bit era zero, e in PORTB abbiamo anche altre periferiche collegate !!!

L'operazione di AND setta o resetta la flag Z nel registro STATUS corrispondentemente al risultato. La riga 11 controlla il risultato e decide di uscire dalla routine solo se il risultato dell'AND è zero ovvero se i pulsanti sono tutti rilasciati.

In questo modo alla prima pressione di un qualsiasi pulsante, la routine Button effettua l'operazione richiesta (inc dec o azzeramento) e poi ne attende il rilascio prima di proseguire.

DispUpdate

```
1     bsf PORTB,2           ;accende led 2 di debug

2     movf Counter,w   ;preleva contatore attuale
3     subwf CounterOld,w    ;sottraigli il valore del loop precedente
4     btfsc STATUS,Z   ;controlla se =0 cioè se valori ancora uguali
5     goto DispUpdateEnd    ;esci se i valori sono uguali

6     movlw 8ch        ;comando posizione cursore sul numero da aggiornare
7     call DispC       ;invia comando al display

8     swapf Counter,w  ;preleva il nibble alto in quello basso di W
9     andlw 0fh        ;maschera i bit che non servono
10    call Hex2asc     ;converti in ascii visualizzabile
11    call DispD       ;mostra a display

12    movlw 0fh        ;maschera i bit che non servono
13    andwf Counter,w
```

```
14    call Hex2asc     ;converti in ascii visualizzabile
15    call DispD       ;mostra a display

16    movf Counter,w   ;aggiorna il valore di CounterOld al nuovo valore
17    movwf CounterOld

DispUpdateEnd

18    bcf PORTB,2           ;spegne led 2 di debug

19    return
```

Questa routine si occupa di aggiornare il valore di *Counter* mostrato sul display.
Per evitare sfarfallii sul display ne aggiorniamo il valore solamente se *Counter* cambia.
Per capire se il valore cambia usiamo una memoria aggiuntiva: *CounterOld* .
Ogni volta che aggiorniamo il display con un nuovo valore nel contempo aggiorniamo anche questa memoria. Con una semplice sottrazione fra i due valori possiamo vedere nei loop successivi se il valore di *Counter* è cambiato oppure è rimasto lo stesso dell'ultimo aggiornamento.
Le righe 2, 3 e 4 effettuano appunto questa operazione.
Prima preleviamo *Counter* in W e poi lo sottraiamo a *CounterOld* facendo ben attenzione che il risultato dell'operazione matematica rimanga in W (flag '*,w'*).
Nella riga 4 verifichiamo se il valore risultante è zero nel qual caso con la riga 5 usciamo direttamente dalla subroutine.

Si potrebbe obbiettare che al posto della *goto* si poteva usare direttamente una *return*.
Come regola generale vi sconsiglio di uscire da una subroutine in più punti.
Supponiamo di modificare il programma in modo tale che prima di uscire da una qualche subroutine debba effettuare qualche operazione di ripristino variabili ... supponiamo anche che la routine sia piuttosto lunga... se dall'inizio avete sparso qua e là delle *return* potreste incappare in problemi poi difficili da trovare !
Prendete sempre l'abitudine di uscire da una subroutine in un punto soltanto ponendovi cioè una sola *return* preceduta da una label con il nome della routine seguito dal suffisso *End.*
Se dovete uscire prematuramente da una subroutine vi basterà una *goto xxxEnd* .
Renderete molto più pulito il programma e vi cautelerete da problemi di difficile debug.
Ovviamente potete fare eccezioni se la routine è molto corta !

Torniamo alla nostra routine di aggiornamento display.

Se l'esecuzione procede alla riga 6 vuol dire che il valore di *Counter* è cambiato ed il display va aggiornato.
Per prima cosa posizioniamo il cursore subito dopo la scritta 'Contatore: ' che avevamo creato durante l'inizializzazione del programma. Facciamo questo calcolando che la scritta di cui sopra occupa 11 caratteri a partire dalla locazione zero che è la prima in alto a sinistra del display.
Dobbiamo quindi posizionare il cursore del display alla posizione 12 e lo facciamo mediante il comando 80h del display cui va aggiunta la posizione richiesta, che nel nostro caso è 0ch, ottenendo quindi 8ch.
Inviamo poi questo comando al display con una chiamata a *DispC* alla riga 7.

Facciamo ora qualche considerazione su cosa visualizzare a display.

Nella memoria RAM del nostro processore abbiamo un solo byte che contiene *Counter* .
Se lo inviassimo tale e quale al nostro display otterremmo qualcosa di difficilmente comprensibile.
Questo perché il display interpreta e visualizza solo caratteri codificati in ascii dove il numero zero ad esempio è codificato come 48...
Supponiamo che *Counter* valga 113 ...
Per visualizzarlo correttamente non abbiamo bisogno di una cifra soltanto... ma di tre !
Questo perché noi siamo abituati a visualizzare i numeri in base 10.
Nella prossima lezione effettueremo tutti i passi necessari per visualizzare in un modo amichevole numeri in base 10... per ora ci limiteremo a convertirli in esadecimale.
La scelta della notazione in base 16 comporta una semplificazione notevole del programma.
Anzitutto perché visualizzare un numero ad 8 bit in esadecimale comporta un uso fisso di cifre: 2
Ogni cifra può avere un valore compreso fra 0 e 16 cioè 0 e F, e ciascuna cifra rappresenta un nibble della memoria ad 8 bit. Con una operazione di *swapf* possiamo facilmente prelevare il nibble che ci serve da *Counter* e convertirlo in ascii con una apposita routine.

La riga 8 effettua appunto questa operazione, preleva il nibble alto da *Counter* in W.
La successiva riga 9 maschera i bit che non servono, cioè il nibble alto (che era il nibble basso di *Counter*). Ora in W abbiamo un numero compreso fra 0 ed F .
Hex2asc alla riga 10 converte questo numero in un carattere stampabile codificato in ASCII che alla riga 11 inviamo al display.
Precisiamo che *Hex2asc* converte in ascii solo numeri compresi da 0 ad F cioè dobbiamo avere in W il nibble alto comunque a zero !

Abbiamo così visualizzato la parte alta di *Counter*, ci resta da processare quella bassa ed è quanto facciamo nelle righe 12/15 in cui semplicemente mascheriamo il nibble alto di *Counter* ed inviamo il risultato al display. Essendo questa la seconda scrittura che effettuiamo sul display, essa risulterà più a destra dato che il cursore dell'LCD incrementa automaticamente ed avremo quindi rispettato l'ordine di posizionamento dei due nibble, prima il più significativo e poi quello di ordine più basso.

Dopo aver aggiornato il display dobbiamo necessariamente aggiornare anche *CounterOld* per impedire il successivo riaggiornamento al prossimo passaggio del programma, a meno che *Counter* non sia cambiato...
E' quello che facciamo alle righe 16 e 17.

Resta solo da spegnere il led di debug acceso all'inizio della routine.
Il tempo di esecuzione di *DispUpdate* è dell'ordine di 200-300 microsecondi, ma dato che il richiamo è continuo ogni 20 millisecondi (tempo scandito dal Loop principale) dovremo, se tutto funziona correttamente, vedere acceso debolmente il led 2. Importante la sua funzione durante il debug.

Hex2asc

```
1     movwf X2asctmp
2     sublw 9           ;guarda se > 9
3     btfss STATUS,C
4     goto  Hex2asc1    ;oltre se lo e'
5     movf  X2asctmp,w  ;recupera il dato
6     addlw 30h         ;somma 30 per trasformarlo in ascii
```

```
7     return

Hex2asc1
8     movf  X2asctmp,w ;recupera il dato
9     addlw 37h        ;somma 37h per avere da A ascii in poi (41h...)
10    return
```

Questa routine si occupa di convertire un numero compreso fra 0 ed F in un carattere stampabile ASCII.
Per visualizzarlo correttamente dobbiamo mostrare un numero nel range di valori 0-9 o una lettera A/F nel range 10-15.

Per capire come funziona la conversione ci occorre anzitutto dare uno sguardo alla tabella ascii riportata più sopra nella sezione descrittiva del display.

Come purtroppo possiamo subito notare numeri e lettere non sono consecutivi.
Quindi per prima cosa controlliamo se il numero passato è nel primo range 0-9 nel qual caso dovremo mostrare un numero o se ricade nel secondo range 10-15 per cui sarà necessaria una lettera.

Alla riga 1 salviamo il valore passato alla subroutine in una locazione temporanea per non rischiare di distruggerlo con operazioni matematiche.

Per controllare se un numero è maggiore o minore di un'altro occorre fare una sottrazione con una costante. Nella stramba architettura del pic durante l'esecuzione della sottrazione se il risultato è minore di zero verrà generato un riporto e verrà resettata a zero una flag di CARRY nel registro STATUS.
Se CARRY è invece settata ad uno allora NON si è generato alcun riporto.
Basterà testare questa flag per stabilire se abbiamo generato un riporto e quindi il numero originario era minore della costante sottratta o se NON è stato generato alcun riporto e quindi il numero originario era uguale alla costante sottratta (risultato zero) o maggiore di essa.

Alla riga 2 quindi sottraiamo al valore costante 9 il conenuto di W.
ATTENZIONE: è W che viene sottratto al valore 9 e non il valore 9 a W !!!
Se NON si genera un riporto (flag CARRY=1) vuol dire che '9 - valore' > o = a zero e quindi valore è compreso fra 0 e 9.
Se si genera riporto (Flag carry = 0) vuol dire che '9-valore' < di zero quindi valore è superiore a 9

Saltiamo a `Hex2asc1` se il valore di W era maggiore di 9 quindi dovremo convertirlo in una lettera, altrimenti proseguiamo con la riga 5 e convertiamo il risultato in un numero ascii.

Alla riga 5 recuperiamo il valore che la *subwf* ha corrotto e che avevamo salvato in *`X2asctmp.`*
Questo deve ora essere compreso fra 0 e 9, la codifica ascii prevede che la cifra zero sia rappresentata dal numero 48 poi gli altri numeri seguono.
Basterà allora sommare 48 al contenuto di W per ottenerne la conversione e poi uscire dalla routine.

Nel caso delle lettere, dalla riga 8 in poi, vediamo che in ascii la lettera A viene codificata dal numero 41h e poi le successive a seguire.
Basterebbe sommare 41h ?

No... perchè quando vogliamo visualizzare la lettera A il valore che abbiamo già in W è 10, per visualizzare B abbiamo 11 e così via...
Da 41h dobbiamo quindi sottrarre 10 che fa 37h.
Facciamo un po di conti allora:

W	*costante*	*risultato della somma*	*carattere ASCII*
10	37h	41h	A
11	37h	42h	B
12	37h	43h	C
13	37h	44h	D
14	37h	45h	E
15	37h	46h	F

Sembra che funzioni !

Tabelle: qualche considerazione

Nella lezione precedente abbiamo visto per la prima volta l'utilizzo delle tabelle di lookup che ci consentono di convertire un valore a partire da una tabella.
Abbiamo incontrato questo sistema di conversione anche in questo programma, e le reincontreremo quasi sempre !
A differenza della lezione precedente però in ogni tabella qui prima della lista di valori abbiamo incontrato un po di codice in più...

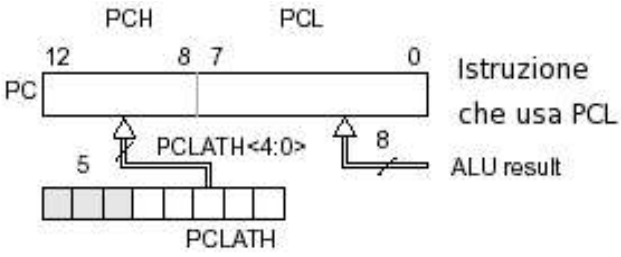
```
1     movwf TabTemp
2     movlw high(TabellaMsg1)
3     movwf PCLATH
4     movf TabTemp,w
5     addwf PCL,f
```

A cosa serve ???

Nell'analisi sull'impiego di PCL nelle tabelle avevamo visto che occorreva essere sicuri di stare lontano dal limite di un blocco di 256 locazioni di memoria programma.
Nel listato della volta scorsa avevamo dislocato la tabella subito dopo il programma. Dato che esso era piuttosto corto eravamo sicuri che tutti i valori della tabella rientrassero sicuramente entro il primo di questi blocchi di memoria da 256 locazioni.

Nel programma di questa lezione invece la lunghezza comincia ad essere notevole...
Magari entro 256 locazioni ci staremo lo stesso... ma è ora di cominciare a cautelarsi da possibili problemi !

Ripassiamo allora il funzionamento di PCL e PCLATH:



Quando si esegue una istruzione che sovrascrive PCL, contemporaneamente all'aggiornamento della parte bassa di PC, la parte alta di quest'ultimo viene presa da PCLATH.
Inutile dire che questo debba essere caricato con un valore corretto.
Nel nostro primo programma non l'abbiamo toccato... all'accensione viene inizializzato a zero, e il nostro precedente, corto, programma risiedeva tutto nel primo blocco di memoria con il risultato che PCLATH=0 andava benissimo.
Ma ora ?

Abbiamo spostato tutte le tabelle nell'ultimo blocco di 256 byte mediante questa direttiva:

```
org 300h          ;ultimo blocco da 256 byte prima di 3FFh che è il
                  ;fine ROM
```

Visualizzare gli indirizzi in esadecimale ci può aiutare a comprendere meglio di cosa stiamo parlando:

numero blocco	***primo indirizzo***	***ultimo indirizzo***	***range PCL***	***PCLATH***
0	0h	FFh	0h/FFh	0
1	100h	1FFh	0h/FFh	1
2	200h	2FFh	0h/FFh	2
3	300h	3FFh	0h/FFh	3

Dopo la *org 300h* ci siamo posizionati nell'ultimo blocco di 256 locazioni memoria programma.
Il numero di blocco e PCLATH vanno di pari passo.
Quindi se scriviamo un valore in PCL, che causerà l'immediato caricamento anche di PCLATH per determinare l'indirizzo di memoria a cui puntare, dobbiamo anche di conseguenza aver precedentemente scritto un valore corretto in PCLATH.

Vediamo allora cosa fanno quelle 5 righe prima di ogni tabella.

La prima semplicemente salva il valore di W passato alla routine, in una locazione temporanea per preservarlo dalla distruzione ad opera delle successive operazione che faranno uso di W.

La successiva :

```
movlw high(TabellaMsg1)
```

carica in w la parte alta (high) dell'indirizzo di memoria della locazione TabellaMsg1.
Quest'ultima altro non è che la locazione iniziale a cui sono memorizzate queste stesse istruzioni.
La parola speciale *high* NON è una istruzione, ma una pseudoistruzione che verrà eseguita

dall'assemblatore nel momento di compilazione del programma.
Letteralmente gli indica di essere sostituita con la parte alta dell'indirizzo dell'etichetta che segue.
Dato che lavoriamo con processori da 8 bit, la parte alta significa il byte più significativo di un valore a 16 bit.
Nel nostro caso, dato che siamo nel terzo blocco di memoria (*org 300h...*) , gli indirizzi saranno nel range 300h/3FFh e quindi *high(TabellaMsg1)* restituirà il valore 3 .

Caricando questo valore in PCLATH alla riga 3 potremo poi con tranquillità eseguire l'istruzione di elaborazione di PCL alla riga 5 (la riga 4 serve solo per recuperare W salvato in precedenza).

Si potrebbe a questo punto obiettare che avendo noi destinato tutte le tabelle all'ultimo blocco di memoria sarebbe bastato comunque precaricare 3 in PCLATH durante l'inizializzazione del programma, dopotutto non si fa altro uso di PCL .
Ma se poi avessimo aggiunto tante tabelle da riempire l'ultimo blocco ?
Saremmo stati costretti a metterne qualcuna nel penultimo !
Cioè *org 200h* .
Il nostro valore 3 di PCLATH non sarebbe più corretto !

Con questo piccolo pezzetto di codice in più ci cauteliamo da qualunque problema, ovunque noi mettiamo le tabelle nella memoria programmi.
L'unica avvertenza che rimane è sempre quella di verificare il non superamento di un blocco qualunque di memoria all'interno di una singola tabella. L'analisi del file con estensione .LST ci permetterà eventualmente di scoprire questo se siamo nel dubbio.