Capitolo I
Della programmazione in C

Struttura del listato

Affrontando la progettazione di un programma in linguaggio di programmazione, occorre ricordare che la chiarezza del codice e la pulizia nella scrittura sono fondamentali, per limitare gli errori e rendere comprensibile il proprio lavoro a chiunque altro vada a verificarlo. Per far ciò, esistono alcune tecniche molto sfruttate dai programmatori scrupolosi.

La principale attenzione del progettista va sicuramente all'*indentazione* del codice, che permette di identificare rapidamente i vari blocchi logici del listato. L'importanza dell'indentazione è evidente nei programmi con cicli annidati, che sarebbero del tutto incomprensibili se scritti tutti con lo stesso inizio di riga.

A titolo di esempio, confrontiamo i seguenti codici:

```
main()
{
char a, b;
for (;;){
if (a!=b){
a=0;
b++;
do{
b=b-a;
}
while(b=-5);
}
}
return 0;
}
```

```
main()
{
   char a, b;
   for (;;){
      if (a!=b){
         a=0;
         b++;
         do{
            b=b-a;
         }
         while (b=-5);
      }
   }
   return 0;
}
```

I due codici sono identici come funzionalità, ma quello di sinistra richiede un'analisi più lunga per capire cosa fa, tanto è nebulosa la sua scrittura. Il listato di destra è invece ben comprensibile. È importante sottolineare come l'operazione di indentazione non sottragga spazio in memoria, quindi non bisogna preoccuparsi in un eventuale bilancio di risorse del sistema.

Capita, poi, che alcune operazioni non siano di immediata comprensione. L'aggiunta di *commenti* è la via ideale per suggerire quello che succede, come visibile nell'esempio proposto.

```
main()
{
          // dichiarazione variabili
          char a, b;

/* inizio operazioni su a e b;
tutto ripetuto all'infinito dentro un for */

          [… programma …]
```

La nota posta prima delle variabili, con una doppia barra, è un *commento a riga singola*. Il commento successivo, invece, essendo compreso tra `/*` e `*/`, è *multilinea*. Compito del

programmatore è porre tali note ovunque sia necessario, nella maniera più chiara possibile, senza però appesantire troppo il codice.

Cicli

Nei codici risolutivi di problematiche relative a sistemi funzionanti senza sosta, tutte le operazioni devono avere la possibilità di essere eseguite sempre. Il metodo impiegato è quello del *ciclo infinito*, che non presenta alcuna condizione di uscita dal ciclo stesso ed è quindi perpetuo.

La realizzazione di un ciclo infinito è così esprimibile, in linguaggio C:

```
main()
{
  // dichiarazione variabili

  for (;;){

    [… programma …]

  }
}
```

Debounce degli ingressi

Si parla di debounce per quei processi eseguiti quando l'ingresso di abilitazione vale 1, ma solo una volta: la successiva esecuzione può avvenire quando lo stesso ingresso è attivo, ma dopo essere stato portato a 0.

Facciamo un esempio più chiaro. Vogliamo che sia incrementata una variabile quando una variabile `pulsante` (che per semplicità vale 1 se spinto e 0 se rilasciato) indica che lo stesso è spinto. Se tenessimo sempre premuto il pulsante, senza il controllo di debounce, la variabile di controllo sarebbe incrementata ad ogni ciclo di clock dell'esecutore (che può essere un PC, un microcontrollore, etc.). Perché, al contrario, avvenga un incremento ad ogni pressione, occorre una variabile che controlli lo stato del debounce.

```
unsigned char flag, cont;

main()
{
  flag = 1;

  for (;;){

    if (pulsante && flag){
      flag = 0;
      cont++;
    }
    if (pulsante == 0) flag = 1;
  }
}
```

Quando, contemporaneamente, `pulsante` e `flag` valgono 1, l'operazione richiesta è eseguita. Siccome dopo la prima esecuzione `flag` viene portato a 0, e rimane tale fino a quando non si rilascia il pulsante, l'operazione è svolta solo una volta per ogni pressione.

L'utilità del debounce degli ingressi sta proprio nell'uso dei pulsanti, che forniscono livelli costantemente attivi per tutta la durata della loro pressione e rischiano di far eseguire migliaia di volte per secondo le istruzioni programmate; con questa costruzione software il problema scompare.

Capitolo II
I PICmicro e il PIC16F84(A)

Caratteristiche dei PICmicro

La Microchip®, ormai nota casa costruttrice americana, da ormai vent'anni è leader nel settore dei piccoli microcontrollori, dedicati all'utenza più inesperta ma da non disdegnare per progetti impegnativi. La serie PIC® (*Peripheral Interface Controller*, controllore a interfaccia periferica) è divenuta celebre per la facilità di programmazione e versatilità di impieghi, sono dunque essi accessibili anche per un uso scolastico didattico.

Caratteristiche fondamentali dei microcontrollori, e dei PICmicro in generale, sono:

– presenza di linee utilizzabili, a seconda della programmazione, come ingressi o come uscite;

– presenza di memoria interna (ROM o FLASH) nella quale memorizzare il software elaborato, di memoria volatile (RAM) per i dati e di unità di elaborazione dati (ALU e/o CPU);

– presenza di sistemi contatori, timer e generatori di clock;

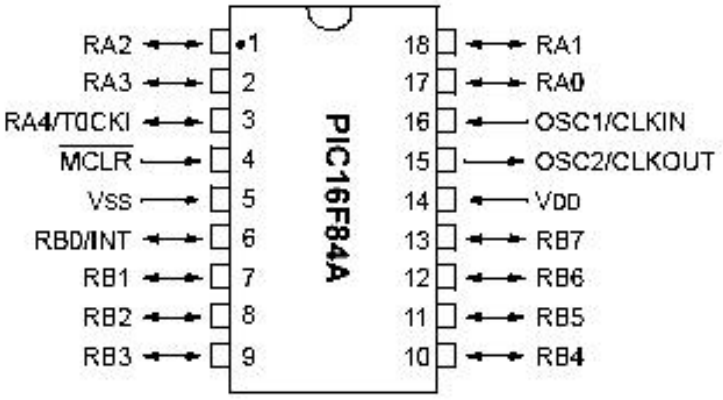
– periferiche di vario tipo per interfacce diverse dai sistemi tradizionali di trasmissione dati e per differenti tipologie di dati.

Nello specifico, i PICmicro sono caratterizzati da un set di istruzioni RISC, molto ridotto perché poche istruzioni sono implementate sul silicio; la velocità è quindi privilegiata, come la facilità nell'apprendimento delle istruzioni.

Per quanto riguarda la costruzione, i PICmicro si presentano come integrati con numero di pin variabile da poche unità ad alcune decine, a seconda delle caratteristiche e delle prestazioni del modello scelto. L'architettura interna non è, come si analizzerà nel capitolo IV, di tipo Von Neumann, ma segue la filosofia Harvard, nella quale la memoria riservata ai dati è completamente separata da quello del programma, permettendo di avere i due accessi separati e migliorando così le prestazioni come velocità di esecuzione.

Struttura del PIC16F84(A)

Il PIC16F84 si presenta come un integrato con 18 pin, in package plastico di tipo DIP.



L'impostazione dei pin è:

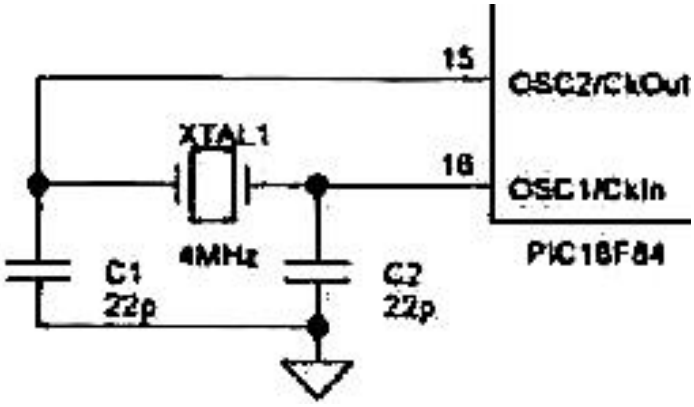
14	alimentazione: + 5 V
5	massa
4	MCLR: abilitazione generale del microprocessore
16, 15	connessioni per l'oscillatore esterno

17, 18, 1–3	porta A: 5 ingressi/uscite programmabili
6–13	porta B: 8 ingressi/uscite programmabili

L'alimentazione deve essere impostata tra 2 V e 6 V, nonostante questo PICmicro lavora in logica TTL.

Un valore logico pari a 0 su MCLR impedisce al PICmicro di funzionare, ponendolo in stato di reset. Se la tensione ad esso applicata è pari, invece, a 12 V, il PICmicro viene programmato.

Perché possano essere impostate temporizzazioni via software, ma anche per il normale funzionamento dell'integrato, occorre progettare una rete oscillatoria con una frequenza che determina il clock di funzionamento. Tale rete può essere composta di una rete RC (economica ma imprecisa) o da una rete con quarzo (meno economica, ma puntuale). Il PIC16F84 ha una frequenza massima di lavoro di 4 MHz (che si riduce a 1 MHz effettivo), mentre nell'evoluzione 16F84A la frequenza effettiva è stata innalzata a 4 MHz.



Il funzionamento delle porte programmabili è sempre in logica TTL e dipende dalla programmazione fatta via software.

Gestione dei PICmicro tramite linguaggio ad alto livello e applicazione al C

Per comandare correttamente un microprocessore sono utilizzabili molteplici linguaggi di programmazione:

– il costruttore prevede una serie (nel PIC16F84 sono 35) di istruzioni in codice assembler, piuttosto "leggero" ma di difficile comprensione;

– con appositi compilatori è possibile utilizzare pressoché qualsiasi codice (BASIC, C, FORTRAN, etc.) con l'ausilio di librerie di adattamento dei valori da linguaggio macchina a quello ad alto livello scelto.

Per ragioni di conoscenze pregresse, si è scelto di programmare in C i PICmicro. A tal scopo viene impiegato un compilatore adatto (per esempio PIC-C Lite) e viene inclusa al programma di gestione la libreria `<pic.h>`.

Gestione di ingressi e uscite. Denominazione

Tale libreria opera l'adattamento dei comandi da assembler a C. In particolare, i pin di ingresso/uscita sono accessibili all'editing operando sul valore logico, quindi 0 o 1, di variabili omonime del nome dei pin (da RA0 a RA4 e da RB0 a RB7).

es. Consideriamo alcune possibili operazioni su RB3

IN/OUT	OPERAZIONI	CODICE C	SIGNIFICATO CIRCUITALE
RB3 definito come ingresso	si verifica il valore in ingresso (confronto)	`RB3==0`	RB3 è stato collegato a massa
		`RB3==1`	RB3 è stato collegato a Vcc
RB3 definito come uscita	si determina il valore in uscita (assegnazione)	`RB3=0`	RB3 ha tensione di uscita nulla
		`RB3=1`	RB3 ha tensione di uscita pari a 5 V

È importante notare che:

– verificando il valore di ingresso, il valore della variabile rimane tale fino a che l'elemento di comando di tale valore è nella condizione corrispondente;

– impostando il valore di uscita, questo rimane tale fino a che, con una successiva istruzione, non si modifica il valore della variabile.

Per poter correttamente gestire ingressi e uscite, occorre progettare attentamente il circuito comprendente il PICmicro perché abbia a disposizione la tensione desiderata in ingresso e possa correttamente utilizzare quella in uscita.

I nomi delle variabili riferite ai pin (sulle quali si può operare solo come indicato in tabella) non sono di tipo comprensibile, perché riferite a elementi standard e non al circuito costruito per l'occasione. Tramite l'operazione

```
#define  NOME  NOMEPIN
```

si può assegnare un nome più utile (NOME) alla variabile NOMEPIN (es. RA1, RB6, PORTB, etc.).

es. Se configuriamo RA2 come ingresso di avvio e RB5 come uscita di pilotaggio di un JFET, potremmo rinominare come segue:

```
#define  AVVIO  RA2  // ingresso
#define  JFET   RB5  // uscita
```

Quest'opera assume maggior significato nel momento in cui le variabili da controllare sono parecchie e con funzioni diverse.

Impostazione di ingressi/uscite

L'impostazione dei pin della porta A e della porta B avviene tramite la direttiva al compilatore `TRISA = 0bXXXXXXXX` e `TRISB = 0bXXXXXXXX`, contenute nella funzione `void init (void)`. Assegnando un valore 0 o 1 ai singoli bit (prima indicati con `X`) si determina la peculiarità dei singoli bit, secondo la seguente tabella:

TRISA	0b	–	–	–	RA4	RA3	RA2	RA1	RA0
TRISB	0b	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

I bit impostati a 1 sono ingressi, quelli a 0 sono uscite.

Gestione delle porte di ingresso/uscita. Denominazione

Soprattutto nel caso in cui tutti i bit di una singola porta siano impostati nello stesso modo (ingressi o uscite), è comoda la possibilità di utilizzare una variabile relativa alla porta nella sua

globalità. In altre parole, le porte del microcontrollore possono essere gestite come variabili a 8 bit, i quali sono i singoli pin costituenti la porta. Nella trasduzione operata dalla `pic.h` sono definite due variabili (*registri*) `PORTA` e `PORTB`, utilizzabili come unsigned char e con la configurazione dei bit analoga a quella nella tabella vista prima.

es. Se `PORTB = 100`, il valore binario corrispondente è:

0 1 1 0 0 1 0 0

La porta B è stata evidentemente configurata con tutti i bit come uscite, quindi le uscite RB2, RB6 e RB7 hanno l'uscita a livello alto, mentre le rimanenti sono a livello basso.

Analogamente a quanto visto per i singoli bit, è possibile rinominare le porte con la direttiva `#define` e un nome utile alla programmazione.

es. Se la porta B gestisce due semafori (con due bit non usati, ma non cambia il risultato) si può fare:

```
#define  SEMAFORI  PORTB  // 6 uscite
```

Variabili (registri)

La compilazione in C per PICmicro prevede l'utilizzo di variabili proprietarie di facile impiego. Tra queste, molto usati sono gli `static bit`, variabili a 1 bit con possibile valore 0 o 1 e relativa "leggerezza" di gestione in memoria. Sarà più chiara l'applicazione al momento di studiare i primi software.

È comunque possibile usare tutti i tipi di variabili previsti dallo standard C, tendendo presente quanto segue:

– le variabili a virgola mobile (`float`, `double`, `long double` e simili) richiedono molta memoria per l'esecuzione e rallentano il programma;

– le variabili strutturate e le matrici potrebbero richiedere un hardware esterno dedicato, quindi non si prestano a tutte le applicazioni.

Ciclicità

Come già accennato, i programmi con interfaccia esterna, per funzionare sempre, richiedono un software a funzionamento continuo, quindi il blocco principale deve essere contenuto in un ciclo infinito. Generalmente, all'interno del `main()`, dopo aver richiamato la funzione `init ()` e l'inizializzazione delle variabili, è presente un `for (;;){` che contiene le istruzioni da eseguire.

Funzionalità accessorie

Dopo il programma principale si esplicitano i sottoprogrammi richiamati (o non) all'interno. Nella `void init (void)`, oltre ai già citati `TRISA` e `TRISB`, sono presenti i registri `OPTION` e `INTCON`, con contenuto a 8 bit esplicitati in formato binario (del tutto analogo ai registri `TRISA` e `TRISB`). Le tabelle seguenti illustrano il significato di ogni singolo bit.

OPTION	0b	RPBU	INTEDG	T0CS	T0SE	PSA	Prescaler bit 2	Prescaler bit 1	Prescaler bit 0
INTCON	0b	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

OPTION		
RPBU	resistori interni di pull-up per la porta B	0 : abilita 1 : disabilita

INTEDG	fronte di lettura dell'interrupt	0 : fronte di discesa 1 : fronte di salita
T0CS	selezione della fonte di interrupt	0 : clock 1 : RA4
T0SE	fronte di incremento del timer con interrupt generato da RA4	0 : fronte di salita di RA4 1 : fronte di discesa RA4
PSA	fonte di prescaler	0 : TMR0 1 : Watch Dog Timer (WDT)
Prescaler bit 2÷0	impostazione prescaler (divisore di frequenza)	8 combinazioni binarie da 000 a 111
INTCON		
GIE	abilitazione generale degli interrupt	0 : disabilita 1 : abilita
EEIE	abilitazione interrupt per scrittura su EEPROM interna	0 : disabilita 1 : abilita
T0IE	abilitazione interrupt su TMR0	0 : disabilita 1 : abilita
INTE	abilitazione interrupt su RB0	0 : disabilita 1 : abilita
RBIE	abilitazione interrupt sulle variazioni di RB4÷RB7	0 : disabilita 1 : abilita
T0IF	flag di interrupt su TMR0	0 : conteggio in corso 1 : overflow avvenuto
INTF	flag di interrupt su RB0	0 : conteggio in corso 1 : overflow avvenuto
RBIF	flag di interrupt sulla variazione di RB4÷RB7	0 : conteggio in corso 1 : overflow avvenuto

L'impostazione dei bit di questi due registri dipende dall'applicazione scelta.

Interrupt e cicli di ritardo. Prescaler e impostazioni di base

In fase di presentazione della famiglia PICmicro si è accennato alla struttura Harvard sulla quale è basata la loro costruzione. La conseguente abilità di poter accedere in tempi diversi a dati e software consente al microprocessore di eseguire due diversi processi quasi in contemporanea: quindi, oltre al programma principale, è possibile eseguire un sottoprogramma che agisce negli stessi tempi dell'applicativo più importante.

Questa opportunità è esprimibile, nei PICmicro, impostando un sottoprogramma, definito di interrupt, che non deve essere richiamato nel `main()` perché agisce autonomamente ad ogni overflow del clock. La dichiarazione in C è la seguente:

```
void interrupt NOME_FUNZIONE (void)
{
```

```
  [azzeramento della indicazione (flag) di interrupt]
  [funzione da sviluppare]
}
```

È fondamentale la rapidità di esecuzione, per non bloccare quella del programma principale.

L'impostazione dei registri INTCON e OPTION deve essere volta a procurarsi una sorgente di clock adatta allo scopo e a determinare i tempi di funzionamento.

Vista l'alta velocità di esecuzione delle istruzioni, occorre un divisore di frequenza che, ovviamente, è presente nella famiglia PICmicro. Il prescaler può essere impostato con 8 differenti divisioni, con le seguenti combinazioni binarie da digitare negli appositi bit di OPTION:

Prescaler	bit 2	bit 1	bit 0
1:2	0	0	0
1:4	0	0	1
1:8	0	1	0
1:16	0	1	1
1:32	1	0	0
1:64	1	0	1
1:128	1	1	0
1:256	1	1	1

Il calcolo per decidere il prescaler è di seguito descritto:

1. convertire il tempo, che desideriamo come ritardo,in microsecondi (μs);
2. dividere tale tempo per il valore massimo del prescaler (256);
3. scegliere il prescaler di valore immediatamente superiore a quello del risultato ottenuto.

es. Vogliamo 65.536 μs di ritardo.

$\frac{65.536}{256} = 256$ quindi scegliamo 1:256 (111).

Il registro TMR0 (Timer 0) è dedicato al conteggio del ritardo (se non viene selezionato il WDT). Quando è stato raggiunto il tempo scelto e impostato, il flag (che è stato scelto in INTCON) viene portato a 1 per segnalare "fine conteggio", da 0 al valore di prescaler deciso. La maggioranza delle volte, però, il valore di prescaler ritorna un tempo troppo elevato per i nostri scopi.

es. 50 ms = 50.000 μs

$\frac{50.000}{256} = 256$ quindi scegliamo 1:256 (111), come con 65 ms.

La soluzione è impostare la variabile `TMR0` (che è già prevista come registro e sulla quale si verifica la temporizzazione) a un valore tale per cui essa conteggi solo il tempo voluto. Questo valore di offset del conteggio è detto *preset* ed è il risultato dell'operazione:

$$\text{preset} = \text{prescaler deciso} - \frac{\text{tempo in } \mu s}{\text{prescaler deciso}}$$

Nell'esempio precedente, il valore è 256 – 195 = 61. All'inizio del `main()`, dopo la dichiarazione di `init ()`, va dichiarato `TMR0 = 61`. Il conteggio avviene così da 61 a 256, per un tempo di 50 ms. Se, al contrario, si imposta `TMR0 = -195`, il timer conterà da –195 a 0, con analogo risultato.

Se il tempo di scansione risulta troppo elevato per il prescaler (che non può contare oltre 65 ms), occorre dividere il tempo in tanti piccoli conteggi uguali e fattibili dal `TMR0`.

es. Si vuole un ritardo di 2 secondi.
2 s = 2.000.000 μs = 50.000 μs · 40 (volte)
Come visto, 50 ms di ritardo richiede l'impostazione 1:256 del prescaler e `TMR0 = -195` (o `TMR0 = 61`, a seconda dei gusti). Con le dichiarazioni come detto in precedenza, al momento del ritardo si imposta un ciclo che incrementa per 40 volte una variabile (con un incremento ogni 50 ms): quando la variabile sarà uguale a 40 saranno passati due secondi.

```
unsigned char cont=0;

main()
{
  init ();
  TMR0 = 61;              // e prescaler 1:256, in OPTION
  [altre variabili]

  for (;;){
    if (T0IF){            // si sceglie di usare T0IF
      T0IF = 0;
      cont++;
      if (cont == 40){    // si entra ogni 2 secondi
        cont = 0;
        [...]
      }
    }
  }
}
```

L'azzeramento dei notificatori di conteggio avvenuto (cioè i flag di interrupt, come T0IF, o, come sopra, `cont`) permette, al ciclo successivo, di poter evidenziare ancora una volta il ritardo creato.

Capitolo III
Esempi di applicazioni con il PIC16F84(A)

Esempio di applicazione circuitale

Come già detto, infinite possono essere le soluzioni possibili con un microcontrollore. Lo schema proposto è quello di una semplice scheda di prova per PICmicro, provvista di tre ingressi e dieci linee di uscita, così distribuiti:

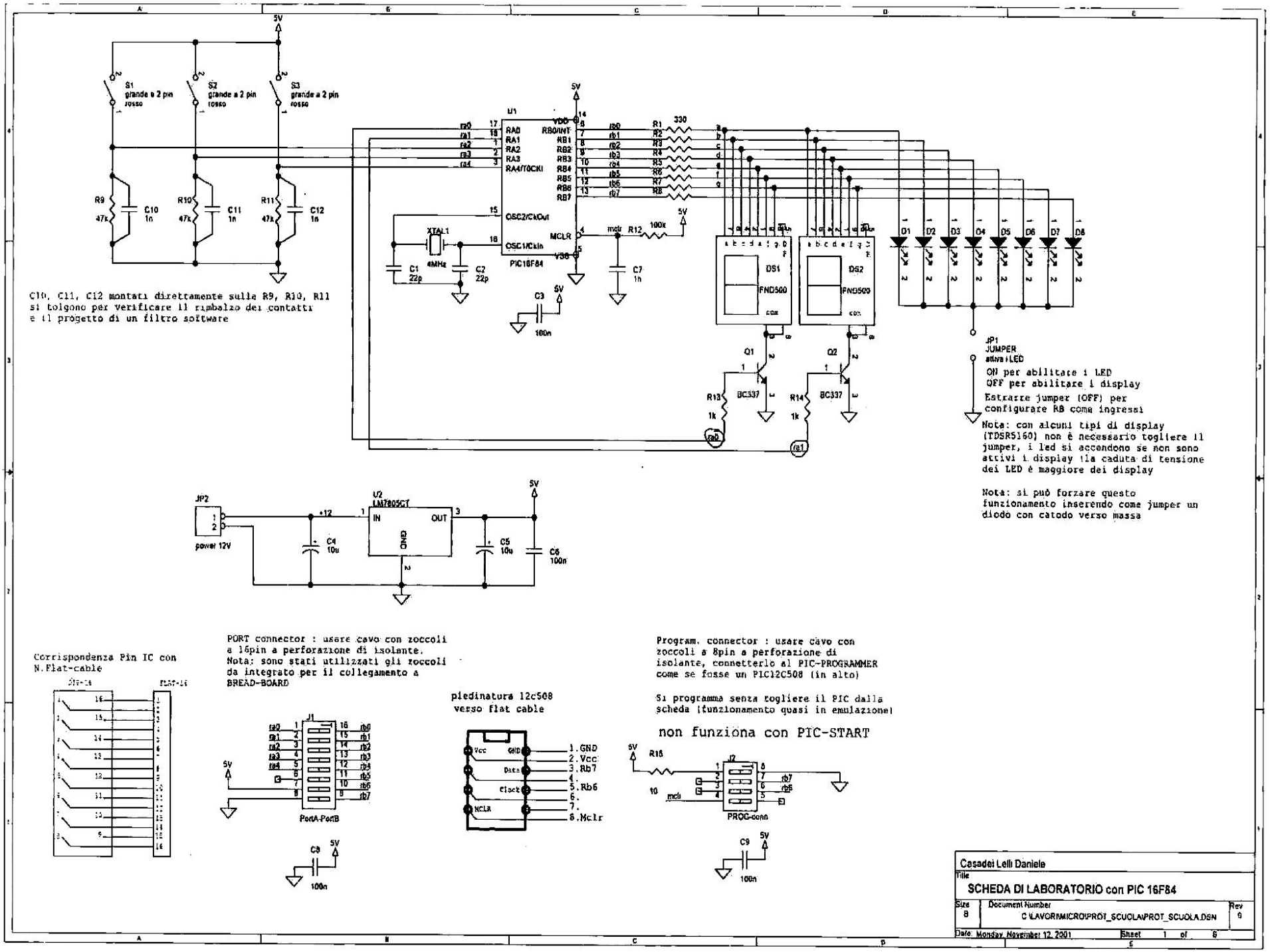
INPUT	RA2, RA3, RA4	pulsanti Normalmente Aperti
OUTPUT	RA0, RA1	abilitazione catodi display 7 segmenti
	RB0 ÷ RB7	LED / anodi display 7 segmenti

I pulsanti, quando azionati, forniscono un livello alto sull'ingresso relativo.

Le abilitazioni dei display avvengono fornendo un livello alto sulle uscite designate, così da porre in saturazione il BJT di comando e permettere la connessione a massa del catodo del display abilitato (e, implicitamente, il funzionamento del display). I livelli alti sulla porta B sono inviati direttamente ai LED, ma giungono comunque anche ai display, nonostante vengano visualizzati solo con l'abilitazione. La visualizzazione su display passa attraverso una codifica dipendente dal peso in binario di ciascun segmento (non è, in altre parole, visualizzato 3 se in uscita dalla porta B c'è il valore binario 3; bisogna "creare" l'uscita binaria corrispondente a 3 sulla porta B).

L'alimentazione della scheda è ottenuta stabilizzando quella di un alimentatore da parete per avere il valore di 5 V: ricordiamo infatti che sono necessari valori TTL per il funzionamento dei PICmicro.

Le connessioni presenti per la programmazione via PC fanno sì che il sistemi funzioni programmando direttamente il PIC16F84(A) sulla scheda, senza bisogno di uno specifico programmatore (*programmazione in-circuit*).


S1 grande a 2 pin rosso
S2 grande a 2 pin rosso
S3 grande a 2 pin rosso
R9 47k
R10 47k
R11 47k
C10 1n
C11 1n
C12 1n
C10, C11, C12 montati direttamente sulle R9, R10, R11 si tolgono per verificare il rimbalzo dei contatti e il progetto di un filtro software
U1
PIC16F84
XTAL1 4MHz
C1 22p
C2 22p
C3 100n
R12 100k
C7 1n
R1 330
DS1 FND500
DS2 FND500
D1 D2 D3 D4 D5 D6 D7 D8
Q1 BC337
Q2 BC337
R13 1k
R14 1k
JP1 JUMPER attiva i LED
ON per abilitare i LED
OFF per abilitare i display
Estrarre jumper (OFF) per configurare RB come ingressi
Nota: con alcuni tipi di display (TDSR5160) non è necessario togliere il jumper, i led si accendono se non sono attivi i display (la caduta di tensione dei LED è maggiore dei display
Nota: si può forzare questo funzionamento inserendo come jumper un diodo con catodo verso massa
JP2 power 12V
U2 LM7805CT IN OUT GND
C4 10u
C5 10u
C6 100n
Corrispondenza Pin IC con N.Flat-cable
PORT connector : usare cavo con zoccoli a 16pin a perforazione di isolante.
Nota: sono stati utilizzati gli zoccoli da integrato per il collegamento a BREAD-BOARD
J1
PortA-PortB
C8 100n
piedinatura 12c508 verso flat cable
1.GND
2.Vcc
3.Rb7
4.
5.Rb6
6.
7.
8.Mclr
Program. connector : usare cavo con zoccoli a 8pin a perforazione di isolante, connetterlo al PIC-PROGRAMMER come se fosse un PIC12C508 (in alto)
Si programma senza togliere il PIC dalla scheda (funzionamento quasi in emulazione)
non funziona con PIC-START
R15 10
J2
PROG-conn
C9 100n
Casadei Lelli Daniele
Title
SCHEDA DI LABORATORIO con PIC 16F84
Size B
Document Number C:\LAVORI\MICRO\PROT_SCUOLA\PROT_SCUOLA.DSN
Rev 0
Date: Monday, November 12, 2001
Sheet 1 of 6

Listato generico

Lo studio del software di comando per il circuito visto riguarda essenzialmente il programma principale, quello contenuto nel `main()`. Il codice restante, con la dichiarazione di sottoprogrammi, configurazioni e intestazioni, è pressoché costante e subisce pochissime sporadiche variazioni (che saranno comunque scrupolosamente indicate).

Ecco di seguito il listato sul quale opereremo.

```
/* Nome_programma.c
 * Data:
 * Microprocessore: PIC16F84(A)
 * Hardware: scheda prot_scuola
 ! Compilare con PIC-lite, opzioni -G -O -Zg -FAKELOCAL */

/* corrispondenza fra indirizzi mnemonici e hardware dei   registri
*/
#include <pic.h>

/* prototipi delle funzioni usate */
void init (void);

/* DIRETTIVE AL COMPILATORE */
__CONFIG (0011111111110001B);
//          __________||++--cristallo
//               |    |+---- WDT 1 = enable
//               |    +----- Power-up timer
//               +---------- CP code prot

#define EnDecine RA0
#define EnUnita  RA1
#define Puls1    RA2 // ingresso
#define Puls2    RA3 // ingresso
#define Puls3    RA4 // ingresso

#define Led1     RB0
#define Led2     RB1
#define Led3     RB2
#define Led4     RB3
#define Led5     RB4
#define Led6     RB5
#define Led7     RB6
#define Led8     RB7

#define Display  PORTB // byte

/* variabili */

/* BIT */

/* vettori */

/* tabelle in ROM */

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  [programma da sviluppare]
```

```
}

void init (void)
{
  TRISA = 0b11111100 ;  // PORT A
  TRISB = 0b00000000 ;  // PORT B

  OPTION = 0b10000000;  // hardware
  INTCON = 0b00000000;  // interrupt
}
```

Programmi con comandi esterni

Accensione e spegnimento LED

Vogliamo che un LED venga acceso alla pressione di un pulsante e spento alla pressione di un secondo pulsante.

```
// Comanda_LED.c
// Data: 26/05/2005

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;

  for (;;){

    if (Puls1){  // accensione LED
      Led1 = 1;
    }

    if (Puls2){  // spegnimento LED
      Led1 = 0;
    }
  }
}
```

Per ogni ciclo del programma, quando il pulsante 1 è premuto (ma anche dopo, finché non è premuto Puls2) il LED 1 si accende e resta acceso, fino a quando non c' è la pressione di Puls2.

Cambio di stato del LED

Vogliamo che un LED si accenda a una prima pressione di un pulsante e che poi, ripremendo lo stesso pulsante una seconda volta, si spenga.

È possibile una variante in cui siano due i pulsanti: uno accende e l'altro spegne il LED. In questo caso non serve il debounce, perché ogni pulsante svolge solo una funzione.

```
// Stato_LED.c
// Data: 26/05/2005

/* BIT */
static bit fPuls1 = 1;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;

  for (;;){
```

```
    if (Puls1 && fPuls1){
      fPuls1 = 0;
      Led1 = !Led1;
    }
    if (Puls1 = 0){
      fPuls1 = 1;
    }
  }
}
```

Lampeggio LED

Vogliamo che, tenendo premuto un pulsante, cominci a lampeggiare un LED con temporizzazione fissata via software. Il valore prescelto è 65 millisecondi (prescaler 1:256; preset = 0).

```
// Lampeggio_LED.c
// Data: 26/05/2005

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;

  for (;;){
    if (Puls1){
      if (T0IF){
        T0IF = 0;
        Led1 = !Led1;
      }
    }
  }
}

void init (void)
{
  TRISA = 0b11111100 ;
  TRISB = 0b00000000 ;

  OPTION = 0b10000111;
  INTCON = 0b00100000;
}
```

Notiamo qui come il debounce sia del tutto inutile, in quanto è il software a gestire le operazioni sul LED tramite la temporizzazione.

Scorrimento LED con inizio costante

Vogliamo che, su una barra di LED, se ne accenda uno solo per volta in modo da ottenere un effetto scorrimento. Questo deve cominciare sempre dallo stesso capo della barra, anche a scorrimento finito. Due versioni con pulsante sempre attivo o singola pressione.

```
// Scorrimento_LED.c
// Data: 26/05/2005

/* BIT */
static bit fPuls1 = 1;

void main (void)  // PROGRAMMA PRINCIPALE
{
```

```
  init();
  Display = 1;

  for (;;){

    if (Puls1 && fPuls1){  // uno scorrimento per pressione
      fPuls1 = 0;
      Display <<= 1;
      if (Display == 0) Display = 1;
    }

    if (Puls1 = 0){
      fPuls1 = 1;
    }
}
```

Evidente il debounce che impedisce, tenendo premuto il pulsante, il velocissimo scorrimento dei LED.

Segue la versione con uno scorrimento temporizzato.

```
// Scorrimento_LED_temp.c
// Data: 26/05/2005

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 1;

  for (;;){

    if (Puls1){  // scorrimento temporizzato (65 ms)
      if (T0IF){
        T0IF = 0;
        Display <<= 1;
        if (Display == 0) Display = 1;
      }
    }
}

void init (void)
{
  TRISA = 0b11111100 ;
  TRISB = 0b00000000 ;

  OPTION = 0b10000111;
  INTCON = 0b00100000;
}
```

Sempre la gestione via software consente, tenendo premuto il pulsante, che in automatico avvenga lo scorrimento.

Scorrimento LED avanti-indietro

Vogliamo che, su una barra di LED, se ne accenda uno solo per volta, in modo da ottenere un effetto scorrimento. Il LED acceso deve “invertire” il proprio senso di marcia quando raggiunge un estremo della barra. Due versioni con pulsante sempre attivo o singola pressione.

```
// Avanti_indietro.c
// Data: 26/05/2005

/* variabili */
unsigned char avanti = 1;
```

```
/* BIT */
static bit fPuls1;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 1;

  for (;;){

    if (Puls1 && fPuls1){  // uno scorrimento per pressione
      fPuls1 = 0;
      if (avanti){
        Display <<= 1;
        if (Display == 0){
          Display = 128;
          avanti = 0;
        }
      }
      if (avanti == 0){
        Display >>= 1;
        if (Display == 0){
          Display = 1;
          avanti = 1;
        }
      }
    }

    if (!Puls1) fPuls1 = 1;
  }
}
```

Nel programma precedente, come nel seguente, occorre usare una variabile di accesso che determina da quale lato dobbiamo gestire lo scorrimento. La variazione di questa variabile avviene all'interno delle singole selezioni con comando `if`, quando è evidente che lo scorrimento è terminato (`PORTB` assume valore 0).

Segue la versione con scorrimento temporizzato.

```
// Avanti_indietro_temp.c
// Data: 26/05/2005

/* variabili */
unsigned char avanti = 1;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 1;

  for (;;){

    if (Puls1){
      if (T0IF){    // scorrimento ogni 65 ms
        T0IF = 0;
        if (avanti){
          Display <<= 1;
          if (Display == 0){
            Display = 128;
            avanti = 0;
```

```
          }
        }
        if (avanti == 0){
          Display >>= 1;
          if (Display == 0){
            Display = 1;
            avanti = 1;
          }
        }
      }
    }
  }
}

void init (void)
{
  TRISA = 0b11111100 ;
  TRISB = 0b00000000 ;

  OPTION = 0b10000111;
  INTCON = 0b00100000;
}
```

Scontro di LED

Vogliamo che, su una barra di LED, se ne accendano due simmetrici per volta, in modo da ottenere un effetto di doppio scorrimento e di scontro centrale. I LED accesi devono "invertire" il proprio senso di marcia quando raggiungono gli estremi della barra.

```
// Scontro_LED.c
// Data: 29/05/2005

/* variabili */
unsigned char Reg1=1, Reg2=128;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;

  for (;;){

    if (Puls1){
      if (T0IF){
        T0IF = 0;
        Reg1 <<= 1;
        Reg2 >>= 1;
        if (Reg1 == 0){
          Reg1 = 1;
          Reg2 = 128;
        }
        Display = Reg1 | Reg2;
      }
    }
  }
}
```

Il funzionamento è del tutto simile al listato precedentemente proposto, con il raddoppio delle variabili di controllo dello scorrimento (perché ne avvengono due contemporanee e contrarie).

Accensione ritardata e discriminata di LED

Vogliamo che, tenendo premuto un pulsante, dopo un tempo prefissato si accendano i LED corrispondenti. Tali LED devono spegnersi al rilascio del pulsante.

```
// Ritardo.c
// Data: 10/06/2005

/* variabili */
unsigned char i, Timer1, Timer2;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;
  TMR0 = -195;
  Timer1 = 0;
  Timer2 = 0;

  for (;;){

    if (Puls1);
    if (!Puls1){
      Led1=0;
      Led3=0;
      Led5=0;
      Led7=0;
    }

    if (Puls2);
    if (!Puls2){
      Led2=0;
      Led4=0;
      Led6=0;
      Led8=0;
    }
  }
}

void init (void)
{
  TRISA = 0b11111100 ;
  TRISB = 0b00000000 ;

  OPTION = 0b10000111;
  INTCON = 0b00100000;
}

void interrupt Timer (void)
{
  T0IF = 0;
  if (Puls1){
    TMR0 = -195;
    Timer1++;
    if (Timer1 == 40){
      Timer1 = 0;
      Led1;
      Led3;
```

```
      Led5;
      Led7;
    }
  }
  if (Puls2){
    TMR0 = -195;
    Timer2++;
    if (Timer2 == 40){
      Timer2 = 0;
      Led2;
      Led4;
      Led6;
      Led8;
    }
  }
}
```

Addizione-sottrazione unitaria binaria

Vogliamo che premendo un primo pulsante si incrementi un contatore e premendone un secondo lo si decrementi. I pulsanti sono gestiti in debounce (una pressione, un'operazione).

```
// Piu_meno_binario.c
// Data: 29/05/2005

/* variabili */
unsigned char cont = 0;

/* BIT */
static bit fPuls1, fPuls2;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();

  for (;;){

    if (Puls1 && fPuls1){  // addizione
      fPuls1 = 0;
      cont++;
      Display = cont;
    }
    if (!Puls1) fPuls1 = 1;

    if (Puls2 && fPuls2){  // sottrazione
      fPuls2 = 0;
      cont--;
      Display = cont;
    }
    if (!Puls2) fPuls2 = 1;
  }
}
```

Conversione dato binario in esadecimale e visualizzazione su display

Vogliamo che:

– premendo un pulsante si incrementi un contatore;

– premendo un secondo pulsante si visualizzino le "decine" esadecimali del contatore su display;

– premendo un terzo pulsante si visualizzino le “unità” esadecimali del contatore su display;
– quando non sono premuti i pulsanti per la visualizzazione esadecimale, il contatore sia visibile in formato binario.

```
// Bin-Hex_Display.c
// Data: 12/06/2005

/* variabili */
unsigned char cont;

/* BIT */
static bit fPuls1;

/* vettori */
const unsigned char ledtable[]= {63, 6, 91, 79, 102, 109, 125, 7,
127, 111, 119, 124, 57, 94, 121, 113};

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;
  EnUnita = 0;
  EnDecine = 0;

  for (;;){

    if (Puls1 && fPlus1){       // aumento variabile cont
      fPuls1 = 0;
      cont++;
      if (cont > 99) cont = 0;
    }
    if (!Puls1) fPuls1 = 1;

    /* visualizza la parte esadecimale di cont con peso 1 */
    if (Puls2){
      Display = ledtable [cont >> 4];
      EnDecine = 1;
      EnUnita = 0;
    }

    /* visualizza la parte esadecimale di cont con peso 0 */
    if (Puls3){
      Display = ledtable [cont & 0x0F];
      EnDecine = 0;
      EnUnita = 1;
    }

    /* visualizza cont in binario */
    if (!Puls2 && !Puls3){
      Display = 0;
      EnDecine = 0;
      EnUnita = 0;
      Display = cont;
    }
  }
}
```

Il vettore `const unsigned char ledtable` contiene le informazioni relative al contenuto da assegnare a PORTB per ottenere l’accensione dei display a sette segmenti secondo la seguente tabella (determinata sulla base delle connessioni da schema elettrico):

	display 7 segmenti							
dato	g	f	e	d	c	b	a	ledtable [dato]
0	0	1	1	1	1	1	1	32+16+8+4+2+1=63
1	0	0	0	0	1	1	0	4+2=6
2	1	0	1	1	0	1	1	64+16+8+2+1=91
3	1	0	0	1	1	1	1	64+8+4+2+1=79
4	1	1	0	0	1	1	0	64+32+4+2=102
5	1	1	0	1	1	0	1	64+32+8+4+1=109
6	1	1	1	1	1	0	1	64+32+16+8+4+1=125
7	0	0	0	0	1	1	1	4+2+1=7
8	1	1	1	1	1	1	1	64+32+16+8+4+2+1=127
9	1	1	0	1	1	1	1	64+32+8+4+2+1=111
A (10)	1	1	1	0	1	1	1	64+32+16+4+2+1=119
B (11)	1	1	1	1	1	0	0	64+32+16+8+4=124
C (12)	0	1	1	1	0	0	1	32+16+8+1=57
D (13)	1	0	1	1	1	1	0	64+16+8+4+2=94
E (14)	1	1	1	1	0	0	1	64+32+16+8+1=121
F (15)	1	1	1	0	0	0	1	64+32+16+1=113

Assegnando a PORTB il valore del vettore corrispondente a quello istantaneo del dato, viene restituita in uscita la combinazione corrispondente. A causa della connessione dei display occorre un pilotaggio appositamente studiato, come ad esempio un multiplexing; si è, però, preferita la selezione manuale della parte di dato da visualizzare.

Se non vi è scelta (tramite i pulsanti 2 o 3), il dato è visualizzato in binario sugli 8 LED. `dato` può essere incrementato premendo Puls1 (in debounce).

La pressione di Puls2 consente la visione della parte di dato, convertito in esadecimale, con peso 1 sul display di sinistra (decine). La pressione di Puls3 permette, al contrario, di vedere la parte di dato esadecimale con peso 0 sul display di destra (unità).

Le operazioni (su bit) compiute per vedere i due dati sono:

• per le “decine” (che, come le unità, tali non sono, essendo il codice esadecimale) si shifta il corrispondente binario del dato di 4 posizioni verso destra, poiché le 4 posizioni con peso maggiore corrispondono al dato hex con peso 1; venendo inseriti 4 zeri a sinistra, il risultato non cambia;

• per le “unità” si compie l’AND tra i bit del dato e una maschera che annulla i 4 bit di peso maggiore, lasciando i 4 bit minori che sono l’esadecimale a peso 0.

Si può esprimere quanto segue con uno schema.

DATO		
b7 b6 b5 b4	b3 b2 b1 b0	
0 0 0 0	b7 b6 b5 b4	`dato = dato >> 4`
	hex peso 1 (16 valori)	

DATO		
b7 b6 b5 b4	b3 b2 b1 b0	
0 0 0 0	1 1 1 1	`dato = dato & 0x0F`
0 0 0 0	b3 b2 b1 b0	
	hex peso 0 (16 valori)	

In ambedue le situazioni, le componenti esadecimali assumono fino a 16 valori (in fondo sono 4 bit), tanti quanti quelli della tabella di conversione. Quindi si tratta di un ottimo sistema per la visualizzazione su display, che sarà riutilizzato anche con l'aiuto dell'interrupt.

Acquisizione seriale di un byte

Vogliamo caricare un byte in maniera seriale, un bit dopo l'altro a partire dal meno significativo, per mezzo di due pulsanti: uno di scelta tra valore 0 e 1 del bit, l'altro di acquisizione del bit stesso. Un terzo pulsante resetta la configurazione del byte.

```
// Seriale.c
// Data: 10/06/2005

/* variabili */
unsigned char DATO, i, j;

/* BIT */
static bit abilit, input;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;

  for (;;){

    /* reset byte: pressione di Puls3 */

    if (Puls3){
      DATO = 0;
      i = 8;
      abilit = 1;
    }


    /* caricamento seriale byte */
    // si carica il singolo bit premendo Puls2
    // se, in contemporanea, Puls1 è premuto, carica 1
    // altrimenti carica 0

    if (abilit){
      for (j=0;j<i;){
        if (Puls2 && input){
          input = 0;
          DATO = (DATO >> 1) | (0x80 * Puls1);
          j++;
        }
        if (!Puls2){
          input = 1;
        }
```

```
        Display = DATO;  // visualizzazione
        abilit = 0;
      }
    }
  }
}
```

Simulazione encoder

Vogliamo ottenere la simulazione di un encoder:

– premendo un pulsante, o lasciandolo a riposo, si determina se si vuole incrementare o decrementare un contatore;

– premendo un secondo pulsante si svolge l'azione decisa dal primo pulsante.

Entrambi i pulsanti sono gestiti in debounce.

```
// Encoder.c
// Data: 26/05/2005

/* variabili */
unsigned char cont=0;

/* BIT */
static bit fPuls1, fPuls2;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();

  for (;;){

    if (Puls1 && fPuls1){     // rilevamento dato
      fPuls1 = 0;
      if (!Puls2 && fPuls2){  // !Puls2 => avanti
        fPuls2 = 0;
        cont++;
        if (cont < 0) cont = 0;
        Display = cont;
      }
      if (Puls2 && fPuls2){   // Puls2 => indietro
        fPuls2 = 0;
        cont--;
        if (cont > 255) cont = 255;
        Display = cont;
      }
    }

    if (!Puls1) fPuls1 = 1;
    if (!Puls2) fPuls2 = 1;
  }
}
```

Programmi automatici

Lampeggio LED

Vogliamo che un LED cambi stato automaticamente, ad ogni conteggio del `TMR0`.

```
// Lampeggio_automatico.c
// Data: 26/05/2005
```

```
void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 0;

  for (;;){

    if (T0IF){  // lampeggio ogni 65 ms
      T0IF = 0;
      Led1 = !Led1;
    }
  }
}

void init (void)
{
  TRISA = 0b11111100 ;
  TRISB = 0b00000000 ;

  OPTION = 0b10000111;
  INTCON = 0b00100000;
}
```

Scorrimento LED avanti-indietro

Vogliamo che, su una barra di LED, se ne accenda uno solo per volta, in modo da ottenere un effetto scorrimento. Il LED acceso deve "invertire" il proprio senso di marcia quando raggiunge un estremo della barra. Lo scorrimento avviene automaticamente, ad ogni conteggio del TMR0.

```
// Avanti_indietro_automatico.c
// Data: 26/05/2005

/* variabili */
unsigned char avanti = 1;

void main (void)  // PROGRAMMA PRINCIPALE
{
  init();
  Display = 1;

  for (;;){

    if (T0IF){  // scorrimento ogni 65 ms
      T0IF = 0;
      if (avanti){
        Display <<= 1;
        if (Display == 0){
          Display = 128;
          avanti = 0;
        }
      }
      if (avanti == 0){
        Display >>= 1;
        if (Display == 0){
          Display = 1;
          avanti = 1;
        }
```

```
      }
    }
  }
}

void init (void)
{
  TRISA = 0b11111100 ;
  TRISB = 0b00000000 ;

  OPTION = 0b10000111;
  INTCON = 0b00100000;
}
```