Esercitazioni C Enhanced Midrange ECCP/PWM

Avvertenza:

Fino ad ora abbiamo sfruttato le abbondanti risorse della scheda di sviluppo. Però, per procedere con l'analisi delle altre possibilità del PWM, occorre disporre di hardware aggiuntivo.

Per prima cosa, se vogliamo sperimentare le funzioni ECCP dobbiamo impiegare un chip che disponga di questo modulo. I PIC 12F1572 e 16F1619 che abbiamo utilizzato finora non integrano di modulo ECCP/PWM e, di conseguenza, si deve disporre di un altro PIC con questa periferica, ad esempio 16F1824/16F1828/16F1825/16F1829 (ma anche altri chip con ECCP/PWM vanno bene). Per la ricerca delle caratteristiche del microcontroller la via più semplice è quella di utilizzare la pagina di ricerca parametrica del sito di Microchip.

Per sperimentare praticamente le possibilità di controllo con ECCP/PWM descritte negli esempi seguenti **sono necessari componenti esterni alla scheda di sviluppo** secondo la lista seguente:

- **ECCP/PWM singolo con steering**
  nessun componente esterno
- **ECCP/PWM : Comando di un LED bicolore con Half-bridge**
  E' richiesta la disponibilità di un LED bicolore a 2 pin e della relativa resistenza
- **ECCP/PWM : Comando di un motore con Half-bridge**
  E' richiesta la disponibilità di un piccolo motore in cc e di alcuni componenti elettronici
- **ECCP/PWM : Comando di un motore con Full-bridge**
  E' richiesta la disponibilità di un piccolo motore in cc e di alcuni componenti elettronici

16. - Un breve cenno sul PWM di 16F1825/29 e simili.

Vediamo alcune caratteristiche generali per PIC 16F1825/7/9 per quanto riguarda la presente esercitazione. Questi chip dispongono di 4 moduli PWM:

- 1 modulo ECCP/PWM con Full-bridge
- 1 modulo ECCP/PWM con Half-bridge
- 2 moduli CCP/PWM

Il modulo **ECCP/PWM1** Full-bridge avrà a disposizione 4 pin di uscita (**P1A**, **P1B**, **P1C**, **P1D**) mentre il modulo **ECCP/PWM2** Half-bridge disporrà di due uscite (**P2A** e **P2B**).

Alcune uscite sono rilocabili con il registro **APFCON1**. Le assegnazioni sono le seguenti:

pin	default	APFCON1
CCP1/P1A	RC5	-
P1B	RC4	-
P1C	RC3	RC1 - APFCON1<2>=1
P1D	RC2	RC0 - APFCON1<3>=1
CCP2/P2A	RC3	RA5 - APFCON1<0>=1
P2B	RC2	RA4 - APFCON1<1>=1

Essendo sovrapposte varie funzioni, è necessaria una pianificazione precisa nell'impiego dei pin.

Pur esistendo la possibilità di utilizzare **Timer2/4/6**, questi timer, nei chip indicati, sono di **"tipo 2 base"**, ovvero:

- **sono alimentati esclusivamente da Fosc/4**
- **i predivisori sono solo 1:1/1:4/1:16/1:64**
- **non dispongono di funzioni HLT**

Di questo va tenuto conto nella programmazione.

Inoltre è da notare che,per questi microcontroller, **la corrente massima ai pin è il classico 25mA e non 50mA** come per i chip usati nelle precedenti esercitazioni e non esistono pin ad alta corrente.

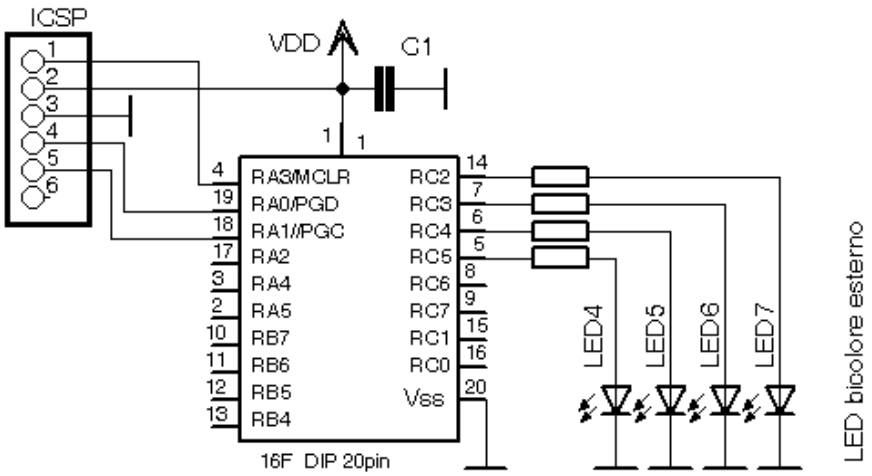
Il chip **non dispone** dell'opzione per variare lo **slew rate** della commutazione sui pin e neppure della possibilità di programmarli come **open drain**.

Anche se gli esempi sono compilati per 16F1829, è possibile utilizzare gli altri chip elencati prima dato che, in relazione ai moduli interessati, hanno la stessa struttura. Basterà cambiare il componente nel progetto e il file di configurazione iniziale.

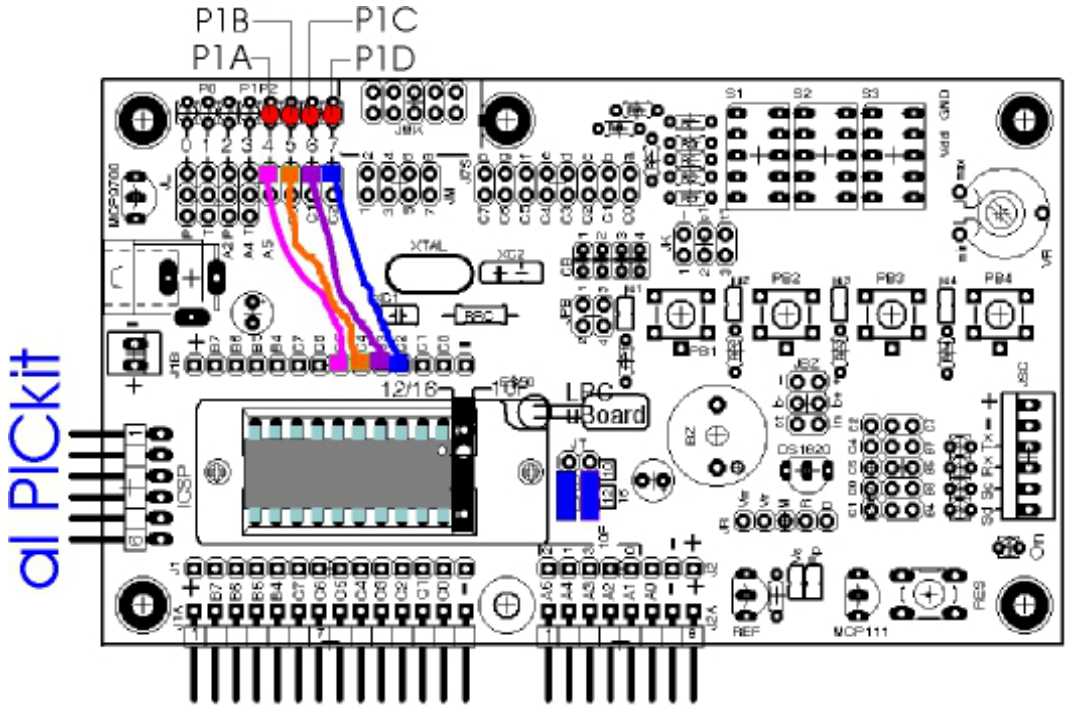
16.1 – PWM singolo con steering – ECCP/PWM1.

Le uscite dei moduli ECCP/PWM , pur essendo 2 o 4, possono essere utilizzate come canale singolo o in steering (con questo termine si indica la possibilità di inviare lo stesso segnale PWM su più pin di uscita contemporaneamente).
Vediamo come sperimentare questa funzione.

Colleghiamo LED ad ognuna delle 4 uscite dell' EPWM.



Sulla scheda di sviluppo:



Le uscite sono dipendenti da un solo modulo e quindi da un solo timer: **avranno tutte lo stesso periodo e duty cycle**, mentre le polarità sono programmabili.

Utilizziamo il modulo **ECCP/PWM1**.
Per prima cosa occorre definire su quali pin abbiamo le uscite. Se non modifichiamo **APFCON1**, abbiamo questa corrispondenza:

pin	default
P1A	RC5
P1B	RC4
P1C	RC3
P1D	RC2

Stabiliamo il timer di supporto: se non operiamo nessuna scelta sul registro **CCPTMRS** abbiamo per default **Timer2**; altrimenti potremo utilizzare **Timer4** o **Timer6**.
In ogni caso, come anticipato, si tratta di timer "**tipo 2 base**", ovvero alimentati solo da **Fosc/4**, con prescaler 1:1, 1:4, 1:16 e 1:64 e senza funzioni HLT.

Lasciamo **Timer2** di default:

```
    //CCPTMRSbits.P1TSEL = 0;     // Select Timer2 - default
```

che è viene programmato per un periodo del PWM di 1ms, dopo aver portato il clock principale a 4MHz. Non c'è una particolare ragione per portare il clock principale a questa frequenza; potremo utilizzare le altre disponibili, adeguando la programmazione dei timer.

Possiamo, ora, impostare una frequenza di PWM di 1kHz, partendo da Fosc/4.

```
// setup Timer2 per 1kHz, clock Fosc/4
void initialize_tmr2(void){
    T2CONbits.T2CKPS = 0b01;     // prescaler 1:4 postscaler 1:1
    PR2 = 250-1;
```

Selezioniamo il modo PWM singolo, creando una funzione di inizializzazione:

```
 // setup ECCP/PWM1
void initialize_epwm1(void){
    CCP1CONbits.P1M = 0b00;      // modo singolo
```

Programmiamo la polarità delle uscite in modo tale da avere le coppie P1A/P1C opposte a P1B/P1D:

```
    CCP1CONbits.CCP1M =0b1101;   // P1A/P1C act. high P1B/P1D act. low
```

Azzeriamo gli LSb del duty cycle, che non useremo:

```
    CCP1CONbits.DC1B = 0b00;     // LSb duty = 0
```

e presettiamo con un duty del 50%:

```
    CCPR1L = 125;                // MSb 50%
```

Non ci interessano le funzioni di auto shutdown e restart, per cui non modifichiamo i default:

```
// CCP1AS = 0;
// PWM1CON = 0;
```

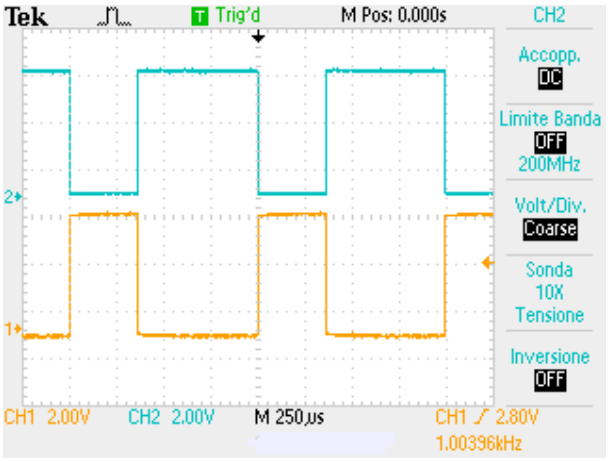
Attiviamo, invece, tutte e quattro uscite di steering:

```
PSTR1CON = 0b00011111;        // steering 4 uscite
```

Il main provvede alle inizializzazioni, dopo di che il loop principale non ha nulla da fare, dato che il modulo **ECCP/PWM** in interrupt esegue la modifica del valore del duty cycle.

L'interrupt dipende dal **Timer2**: dato che il postscaler è 1:1 il flag **IF** andrà a 1 ad ogni fine periodo di 1ms. Contiamo 40 di questi periodi per avere un tempo di 40ms, che è la cadenza di variazione del duty. Il registro **CCPR1L** viene incrementato di 1 ad ogni cadenza e quando arriva a 255 si azzera e riparte. Se vogliamo una cadenza diversa basterà adeguare il limite del conteggio delle interruzioni.

Il risultato è visibile sulle coppie di LED, una che aumenta e l'altra che diminuisce di luminosità fino a che la prima è a piena luminosità e l'altra spenta, per poi scambiare la situazione e riprendere il ciclo.



La forma d'onda in uscita su una coppia di pin.

Sono invertite le durate degli impulsi.

es16C

```
/*****************************************************************
*---------------------------------------------------------------
*    Titolo        :  C Enhanced - es16C
*                     LED comandati da PWM steering con duty cycle
*                     variabile in passi di 40ms.
*                     Impiego del modulo ECCP/PWM1 steering.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  16F1829
*    Autore        :  afg
*
*---------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       16F1829 @ 20 pin
*
*                    __\/__
*              Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*              RC5 -|5   16|- RCO
*              RC4 -|6   15|- RC1
*              RC3 -|7   14|- RC2
*              RC6 -|8   13|- RB4
*              RC7 -|9   12|- RB5
*              RB7 -|10  11|- RB6
*                   |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: In
*    RA4      3: In
*    RA3      4: In
*    RC5      5: Out P1A
*    RC4      6: Out P1B
*    RC3      7: Out P1C
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In
*    RC2     14: Out P1D
*    RC1     15: In
*    RC0     16: In
*    RA2     17: In
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/
```

```
// intosc, no clockout, no wdt, no pwrt, no mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1829.h"

#include <xc.h>

#define _XTAL_FREQ   4000000         // OSCINT 4MHz

//    FUNZIONI.
// clock a 4MHz
void oscint_4M(void){
    OSCCONbits.IRCF = 0b1101;       // intosc 4MHz
}

// inizializza I/O
void initialize_io(void){
    ANSELA = 0;                 // no analogiche
    ANSELB = 0;
    ANSELC = 0;
    TRISC = 0b11000011;         // RA5:2 out
    // abilita wpu per ingressi
    nWPUEN = 0;                 //OPTION_REGbits.nWPUEN = 0
}

// setup Timer2 per 1kHz, clock Fosc/4
void initialize_tmr2(void){
    T2CONbits.T2CKPS = 0b01;    // prescaler 1:4 postscaler 1:1
    PR2 = 250-1;
}
// Timer2 on & clear flag
void tmr2_on(void){
    TMR2IF = 0;
    TMR2ON = 1;
}

 // setup ECCP/PWM1
void initialize_epwm1(void){
    //CCPTMRSbits.P1TSEL = 0;    // Select Timer2
    CCP1CONbits.P1M = 0b00;      // modo singolo
    CCP1CONbits.CCP1M =0b1101;   // PWM mode: PxA/PxC high PxB/PxD low
    CCP1CONbits.DC1B = 0b00;     // LSb duty = 0
    CCPR1L = 125;                // MSb 50%
    //CCP1AS = 0;
    //PWM1CON = 0;
    PSTR1CON = 0b00011111;       // steering
}

// abilita interruzioni
 void enable_irq(void){
    TMR2IF = 0;
    TMR2IE = 1;
    PEIE = 1;
    GIE = 1;
}

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

   oscint_4M();
```

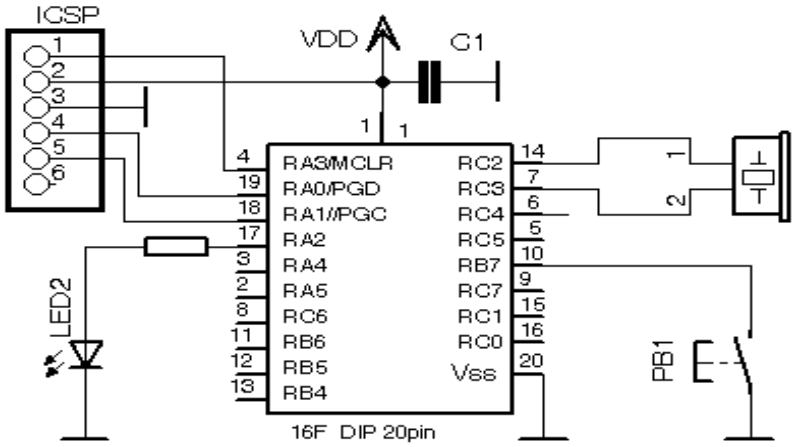
```
   initialize_io();            // Inizializza I/O
   initialize_tmr2();          // setup Timer
   initialize_epwm1();         // setup ECCP
   tmr2_on();                  // timer on
   enable_irq();               // abilita interruzioni

// Loop principale
  while (1){};
  }


/**** Interrupt Routine ****/
void interrupt isr(void){             // C90
//void __interrupt() isr(void){      // C99
static unsigned char  isrcntr = 0;    // contatore passi
    TMR2IF = 0;                       // cancella flag
    isrcntr++;
    if (isrcntr==40){                 // ogni 40ms
      isrcntr = 0;
      CCPR1L += 1;                    // modifica duty cycle
    }
}
```

16.2 – PWM singolo con steering – ECCP/PWM2.

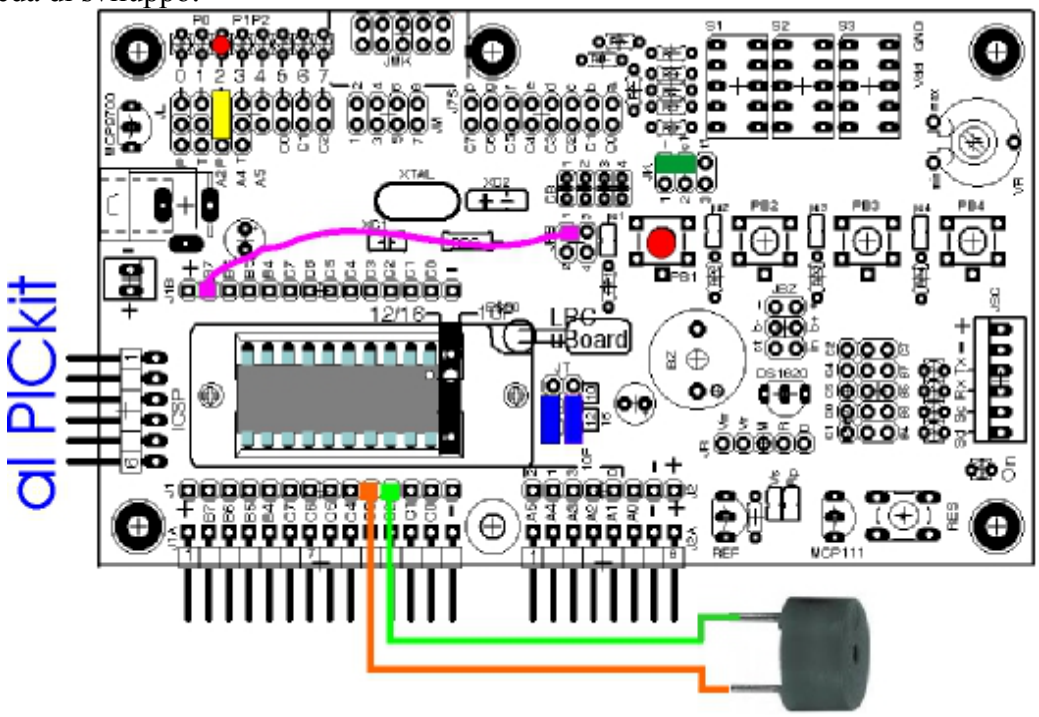
Le uscite del modulo **ECCP/PWM2** sono solo 2. Possiamo usarle in controfase per alimentare un cicalino piezoelettrico, utilizzando lo **steering** con polarità opposte.

Questo tipo di buzzer ha il vantaggio di utilizzare pochissima energia, ma ha anche due punti deboli: il suono è al massimo livello solo per la frequenza di risonanza tipica del componente (generalmente 2-4kHz) e occorre una certa tensione per attivarli in modo utile. Un modello generico, alimentato da impulsi a 5V tenderà ad avere una uscita audio piuttosto bassa; se, però, abbiamo modo di disporre di impulsi ad una tensione maggiore, avremo anche un maggiore volume.

Il suono viene emesso alla pressione di un pulsante, mentre il LED sia accende per indicare l'evento.



Sulla scheda di sviluppo:



Il **buzzer non è polarizzato** (ovvero i due terminali possono essere invertiti senza problemi) e **non richiede resistenze in serie**.

Chiariamo bene che stiamo **non si tratta di un altoparlante elettromeccanico, ma di uno piezioelettrico**; si tratta di due cose ben diverse. L'impedenza del piezoelettrico è elevata, quello dell'altoparlante elettromeccanico è molto bassa e richiede un buffer.

Nell'inizializzazione degli I/O lasciamo come default il **TRISC=1** dove sono applicati i segnali del PWM: questo blocca il segnale in uscita fino a che non porteremo i **TRISC** delle uscite PWM a 0. In sostanza, un interruttore che spegne il segnale al cicalino.

Impostiamo il modulo **ECCP/PWM2** in modo singolo e steering delle due uscite **P2A** e **P2B,** che usiamo con active level opposti e duty cycle 50%.

```
// setup ECCP/PWM2
void initialize_epwm2(void){
    //CCPTMRSbits.P2TSEL = 0;       // Select Timer2 - default
    CCP2CONbits.P2M = 0b00;         // modo singolo
    CCP2CONbits.CCP2M =0b1101;      // PWM mode: PxA active-high PxB
active-low
    CCPR2L = 125;                   // duty 50%
    CCP2CONbits.DC2B = 0b00;
    //CCP2AS = 0;
    //PWM2CON = 0;
    PSTR2CON = 0b00011111;          // steering
}
```

Rilochiamo **P2A** e **P2B** su **RA5** e **RA4** con il registro **APFCON1**

funzione	default	rilocati
CCP2/P2A	RC3	RA5 - APFCON1<0>=1
P2B	RC2	RA4 - APFCON1<1>=1

```
// rilocare P2A e P2B
  APFCON1bits.P2BSEL = 1;    // P2B su RA4
  APFCON1bits.P2ASEL = 1;    // P2A su RA5
```

Il cicalino che abbiamo disponibile ha una frequenza tipica di 4kHz e per questo valore programmiamo il periodo del PWM.

```
// setup Timer2 per 4kHz, clock Fosc/4
void initialize_tmr2(void){
    T2CONbits.T2CKPS = 0b00;     // prescaler 1:1 postscaler 1:1
    PR2 = 250-1;
}
```

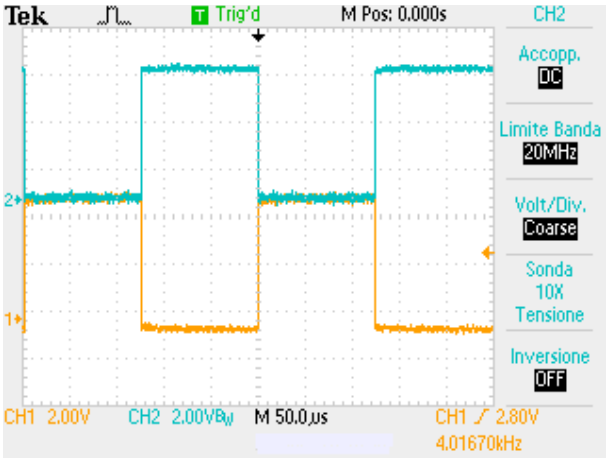
Se avete modelli con diversa frequenza di risonanza, variate di conseguenza il periodo del timer.

Il loop principale verifica lo stato del pulsante. Se è premuto, attiva le uscite PWM e accende un

LED per un minimo di 200ms. Quando viene rilasciato, il LED viene spento e le uscite PWM disabilitate.

```
    while (PB1);                // attesa pulsante premuto
      TRISC = 0b11110011;       // uscita PWM
      buttonled = 1;            // LED acceso
      __delay_ms (200);         // durata minima

    while (!PB1);               // attesa rilascio
      TRISC = 0xFF;             // stop uscita PWM
      buttonled = 0;            // LED spento
    __delay_ms(30);             // debounce
}
```

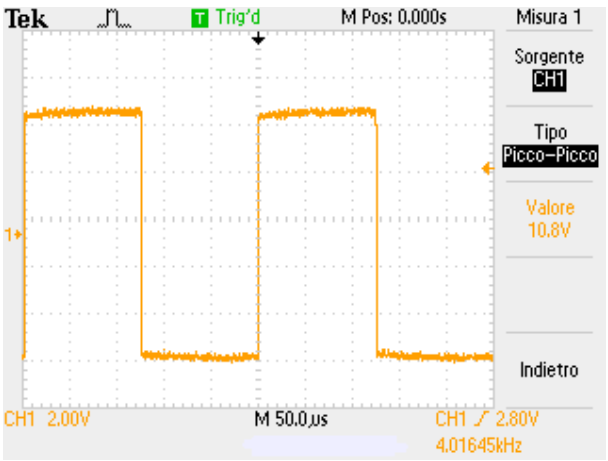
Potete provare a staccare una delle estremità del cicalino dall'uscita PWM e collegarla alla Vss o Vdd: il suono sarà decisamente meno forte. Questo perchè viene alimentato con una transizione da 0 a 5V. Se è collegato alle due uscite PWM che sono in controfase, la tensione relativa ai capi del cicalino sarà doppia e quindi si genererà un suono più deciso.



Le forme d'onda ai pin **P2A** e **P2B**., misurati rispetto alla massa.

I due segnali presentano lo stesso impulso PWM, ma con polarità opposte.

Il duty cycle è al 50% e l'escursione è tra 0 e 5V.



La forma d'onda tra ai capi del buzzer.

Il suo valore assoluto è il doppio della tensione dei singoli impulsi.

es16C_1

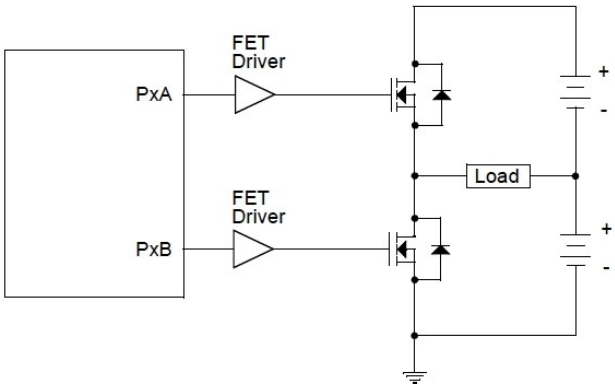
```
/*****************************************************************
*----------------------------------------------------------------
*    Titolo        :  C Enhanced - es16C_1
*                     Buzzer piezo comandato da PWM steering con duty
*                     cycle fisso.
*                     Impiego del modulo ECCP/PWM2.
*                     Uscita dei PWM rilocata con APFCON1.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  16F1829
*    Autore        :  afg
*
*----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ---------------
*       16F1829 @ 20 pin
*
*                    __  __
*                   |  \/  |
*             Vdd -|1    20|- Vss
*       RA5/CLKIN -|2    19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3    18|- RA1/ICSPCLK
*        RA3/MCLR -|4    17|- RA2
*             RC5 -|5    16|- RCO
*             RC4 -|6    15|- RC1
*             RC3 -|7    14|- RC2
*             RC6 -|8    13|- RB4
*             RC7 -|9    12|- RB5
*             RB7 -|10   11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: Out P1A
*    RA4      3: Out P1A
*    RA3      4: MCLR
*    RC5      5: In
*    RC4      6: In
*    RC3      7: In
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In  PB1
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In
*    RC2     14: In
*    RC1     15: In
*    RC0     16: In
*    RA2     17: Out LED
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/
```

```
// intosc, no clockout, no wdt, no pwrt, mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1829.h"

#include <xc.h>

#define _XTAL_FREQ   4000000         // OSCINT 4MHz

#define buttonled  LATAbits.LATA2    // LED di segnalazione
#define PB1        PORTBbits.RB7     // pulsante

//    FUNZIONI.
// clock a 4MHz
void oscint_4M(void){
    OSCCONbits.IRCF = 0b1101;       // intosc 4MHz
}

// inizializza I/O
void initialize_io(void){
    ANSELA = 0;                 // no analogiche
    ANSELB = 0;
    ANSELC = 0;
    LATA = 0;                   // preset led spento
    TRISA = 0b00111011;         // RA2 out
 // abilita wpu per ingressi
    nWPUEN = 0;                 //OPTION_REGbits.nWPUEN = 0
 // rilocare P2A su RA5  P2B su RA4
    APFCON1bits.CCP2SEL = 1;
    APFCON1bits.P2BSEL = 1;
}

// setup Timer2 per 4kHz, clock Fosc/4
void initialize_tmr2(void){
    //T2CLKCON = 0x00;           // T2CS FOSC/4;
    T2CONbits.T2CKPS = 0b00;     // prescaler 1:1 postscaler 1:1
    PR2 = 250-1;
}
// Timer2 on & clear flag
void tmr2_on(void){
    TMR2IF = 0;
    TMR2ON = 1;
}

 // setup ECCP/PWM2
void initialize_epwm2(void){
    //CCPTMRSbits.P2TSEL = 0;       // Select Timer2
    CCP2CONbits.P2M = 0b00;         // modo singolo
      CCP2CONbits.CCP2M =0b1101;    // PWM mode: PxA active-high PxB active-low
    CCPR2L = 125;                   // duty 50%
    CCP2CONbits.DC2B = 0b00;
    //CCP2AS = 0;
    //PWM2CON = 0;
    PSTR2CON = 0b00011111;          // steering
}

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

```
   oscint_4M();
   initialize_io();            // Inizializza I/O
   initialize_tmr2();          // setup Timer
   initialize_epwm2();         // setup ECCP
   tmr2_on();                  // timer on

// Loop principale
  while (1){

    while (PB1);               // attesa pulsante premuto
      TRISC = 0b11110011;      // uscita PWM
      buttonled = 1;           // LED acceso
      __delay_ms (200);        // delay minimo

      while (!PB1);            // attesa rilascio
      TRISC = 0xFF;            // stop uscita PWM
      buttonled = 0;           // LED spento
    __delay_ms(30);            // debounce
  }
}
```

16.3 – Half-bridge – ECCP/PWM2

Un **Half-bridge** (mezzo ponte) è costituito dalla metà di un ponte ad H.



Una applicazione tipica è quella di variare l'energia al carico attraverso il PWM.
Per invertire la polarità occorre disporre di una alimentazione duale.

Il segnale PWM è applicato al carico attraverso un buffer. Quando PxA è attivo, collega il carico alla tensione positiva; quando è attivo PxB lo collega a quella negativa.

Half-bridge può operare sia in cc che ca ed è utilizzato per SMPS, inverter e convertitori vari.

Questa configurazione, però, nasconde un serio rischio per i transistor di potenza: se entrambi sono in conduzione nello stesso momento, ne deriva un corto circuito sull'alimentazione.
Più che a seguito di un errore di comando del programma, la situazione è presente in ogni caso, dato che, in genere, il tempo di accensione di un MOSFET è minore di quello di spegnimento, a causa delle capacità intrinseche presenti. Ne deriva la necessità di inserire un “tempo morto” (dead band) tre lo spegnimento di un MOSFET e l'accensione dell'altro.

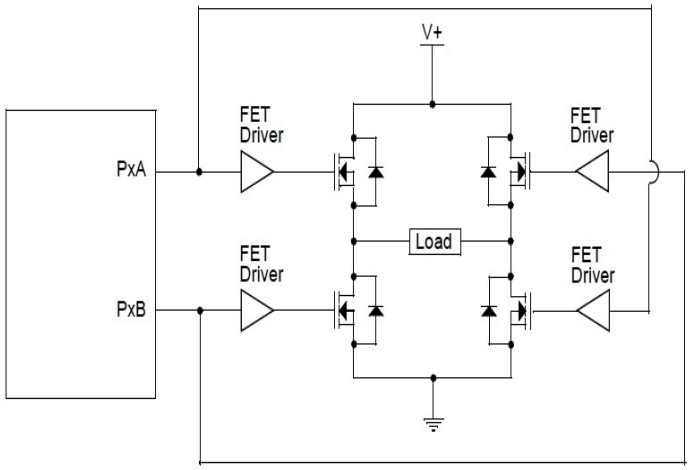
Se vogliamo utilizzare un half-bridge per comandare un motore, dobbiamo considerare che, se il duty cycle è del 50%, il motore sarà alimentato per metà del tempo con una polarità e per l'altra metà con quella opposta, col risultato di rimanere fermo. Aumentando il duty cycle su un transistor, l'altro diminuirà di conseguenza e il motore girerà in una direzione piuttosto che nell'altra.
Qui entrano in gioco l'inerzia del motore e la frequenza del PWM. Se la frequenza è troppo bassa, nella posizione di “fermo” il rotore sarà sottoposto a vibrazioni evidenti; solo per una combinazione giusta tra induttanza del motore e inerzia meccanica il rotore sarà effettivamente fermo.
Quando il duty cycle sarà al 100%, il rotore girerà in una direzione alla massima velocità; con il duty allo 0% sarà lo stesso nella direzione opposta. Tutti i valori di duty cycle intermedi corrisponderanno a velocità diverse.

In una tabella:

Duty cycle	Velocità	Rotazione
0%	massima	inversa
0-50%	in riduzione	inversa
50%	arresto	
50-100%	in accelerazione	diretta
100%	massima	diretta

Avvertenza: **la direzione di rotazione indicata è solo formale**, per distinguere la polarità applicata

al motore, ma non ha nessuna corrispondenza con una rotazione ”inversa” o “diretta” rispetto alle necessità dell'applicazione. Questa corrispondenza dovrà essere creata dagli opportuni collegamenti dei motore al driver.



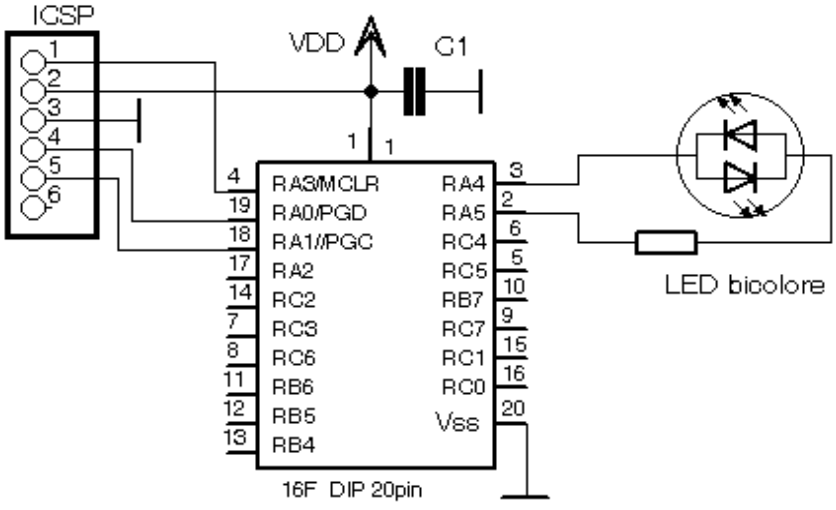
La disponibilità di una tensione duale non è sempre praticabile, per cui possiamo prendere in considerazione una diversa configurazione, ovvero una coppia di Half-bridge, che creano un **H-bridge** comandato da due soli segnali.

PxA comanda il MOSFET superiore a sinistra e quello inferiore a destra, applicando una polarità al motore.
PxB comanda gli altri due MOSFET applicando la polarità opposta.

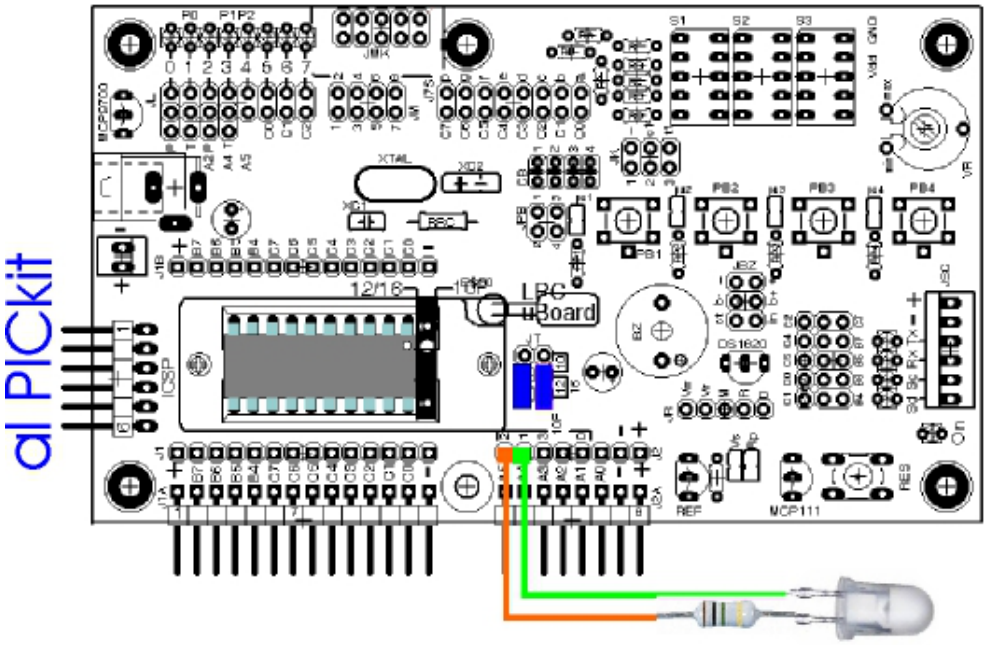
Non si tratta di un Full-bridge, dato che i segnali di comando sono solo due e permane la necessità dell'inserimento della dead band dove sia necessario.

Per realizzare questo ponte, la via più semplice è quella di utilizzare un integrato dedicato, come i classici L293D o SN754410 oppure i più recenti L6393, DRV8801, TB6612, L9110S, ecc.
Questi componenti possono essere assemblati su una breadboard o millefori, ma anche essere acquistati sotto forma di piccoli circuiti stampati già pronti.
In ogni caso, si tratta di aggiungere componenti esterni, necessari in quanto un motore avrà un assorbimento superiore alle possibilità dei pin del microcontroller.
Possiamo, però, verificare il funzionamento dell'Half-bridge anche senza driver esterni : in effetti, un driver puh-pull è già presente dietro il pin e non ne occorre uno esterno se ci limitiamo ad un carico che rientri nei 25mA che i pin di questo chip possono fornire.

Utilizziamo il circuito precedente sostituendo un LED bicolore rosso/verde a due pin al posto del buzzer



Sulla scheda di sviluppo:



Il LED bicolore a due pin non ha polarità: semplicemente, applicando una polarità, si accenderà un colore; invertendo la polarità si accenderà l'altro. La resistenza serie potrà essere 150-270ohm.

Facciamo in modo che il duty cycle vari dallo 0 al 100%. In questo modo il colore del LED passerà da verde a giallo a rosso. Questo perchè con il PWM a segnali opposti, la percentuale del duty cycle determinerà il tempo di accensione dei due colori e quindi il colore risultante dalla miscelazione dei due.

Usiamo un clock a 4Mhz

Nell'inizializzazione dell'I/O rilochiamo **P2A** e **P2B** su **RA5** e **RA4**

```
// rilocare P2A su RA5  P2B su RA4
    APFCON1bits.CCP2SEL = 1;
    APFCON1bits.P2BSEL = 1;
```

Utilizziamo il **Timer6** alimentato a **Fosc/4**, con un prescaler **1:4** e **PR2 = 250-1** genera un periodo di **1ms** (1000Hz).

```
void initialize_tmr6(void){
    T6CONbits.T6CKPS = 0b00;     // prescaler 1:1 postscaler 1:1
    PR6 = 250-1;
```

Inizializziamo il modulo ECCP/PWM2 come Half-bridge

```
    CCPTMRSbits.C2TSEL = 0b10;     // select Timer6
    CCP2CONbits.P2M = 0b10;        // Half-bridge
    CCP2CONbits.DC2B = 0b00;       // LSb duty
    CCP2CONbits.CCP2M = 0b1100;    // PWM PxA/PxB active high
```

Facciamo variare il duty cycle dallo 0% al 100% e viceversa con una certa cadenza.

Il segnale PWM avrà un periodo di 1ms e una variazione di **CCPR2L:DC2B** da 0 a 1000 per il il duty da 0 al 100%.
Possiamo utilizzare tutti e 10 i bit della risoluzione, caricando sia **CCPR2L** che che i due **DC2B**, oppure lasciare gli LSb a 0 e modificare solo gli MSb.

1000 decimale equivale a **1111101000** binario. Il 100% del duty cycle si avrà per:

CCPR2L = 11111010 (250 decimale) e **DC2B = 00**

Per questa applicazione, la risoluzione a 8 bit è più che adeguata, quindi variamo **CCPR2L** da 0 a 250 e poi da 250 a 0 con passi di 40ms.

```
for(dc = 0; dc < 250; dc++){
  CCPR2L = dc;
  __delay_ms(40);
}
for(dc = 249; dc > 0; dc--){
  CCPR2L = dc;
  __delay_ms(40);
}
```

Il LED passerà da verde a giallo a rosso per poi ripetere il ciclo. Se volete variare la velocità del ciclo cambiate il tempo del delay.

Non abbiamo considerata l'inserzione del ritardo di dead band dato che siamo sul push-pull interno al chip che non originano corto circuiti.

es16C_2

```
/*****************************************************************
*----------------------------------------------------------------
*    Titolo        :  C Enhanced - es16C_2
*                     Half-bridge per LED bicolore.
*                     Variazione colore automatica.
*                     Modulo ECCP/PWM2. Rilocazione pin.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  16F1829
*    Autore        :  afg
*
*----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       16F1829 @ 20 pin
*
*                    __  __
*                   |  \/  |
*             Vdd -|1    20|- Vss
*       RA5/CLKIN -|2    19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3    18|- RA1/ICSPCLK
*        RA3/MCLR -|4    17|- RA2
*             RC5 -|5    16|- RCO
*             RC4 -|6    15|- RC1
*             RC3 -|7    14|- RC2
*             RC6 -|8    13|- RB4
*             RC7 -|9    12|- RB5
*             RB7 -|10   11|- RB6
*                  |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: In
*    RA4      3: In
*    RA3      4: MCLR
*    RC5      5: In
*    RC4      6: In
*    RC3      7: Out P2A
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In
*    RC2     14: Out P2B
*    RC1     15: In
*    RC0     16: In
*    RA2     17: In
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/
```

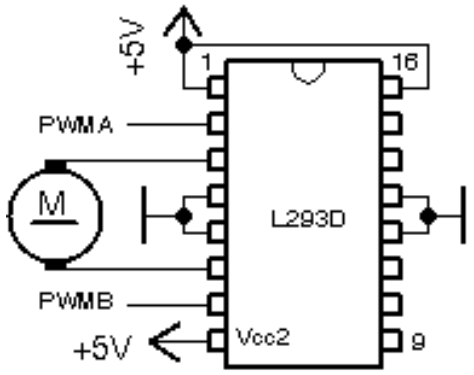
```
// intosc, no clockout, no wdt, no pwrt, mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1829.h"

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ   4000000        // OSCINT 4MHz

uint8_t  dc = 0;                    // valore per duty

//    FUNZIONI.
// clock a 4MHz
void oscint_4M(void){
    OSCCONbits.IRCF = 0b1101;      // intosc 4MHz
}

// inizializza I/O
void initialize_io(void){
    ANSELA = 0;                    // no analogiche
    ANSELB = 0;
    ANSELC = 0;
    TRISA = 0b11001111;            // RA5:4 out
 // abilita wpu per ingressi
    nWPUEN = 0;                    //OPTION_REGbits.nWPUEN = 0
 // rilocare P2A su RA5  P2B su RA4
    APFCON1bits.CCP2SEL = 1;
    APFCON1bits.P2BSEL = 1;
}

// setup Timer6 per 1kHz, clock Fosc/4
void initialize_tmr6(void){
    T6CONbits.T6CKPS = 0b00;    // prescaler 1:1 postscaler 1:1
      PR6 = 250-1;
}
// Timer2 on & clear flag
void tmr6_on(void){
    TMR6IF = 0;
    TMR6ON = 1;
}

 // setup ECCP/PWM2
void initialize_epwm2(void){
    CCPTMRSbits.C2TSEL = 0b10;     // select Timer6
    CCP2CONbits.P2M = 0b10;        // Half-bridge
    CCP2CONbits.CCP2M =0b1100;     // PxA/PxB active-high
    CCPR2L = 125;                  // duty 50%
    CCP2CONbits.DC2B = 0b00;       // LSB non usati
    //CCP1AS = 0;
    //PWM1CON = 0;
    //PSTR2CON = 0;
}

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

   oscint_4M();
   initialize_io();            // Inizializza I/O
   initialize_tmr6();          // setup Timer
```

```
   initialize_epwm2();          // setup ECCP
   tmr6_on();                   // timer on

// Loop principale
  while (1){

    for(dc = 0; dc < 250; dc++){
      CCPR2L = dc;
      __delay_ms(40);
    }
    for(dc = 249; dc > 0; dc--){
      CCPR2L = dc;
     __delay_ms(40);
      }
  }
}
```

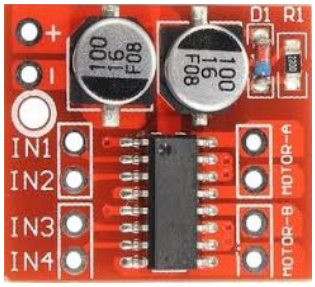
16.4 – Half-bridge per comandare un motore.

Se vogliamo comandare un motore o un LED con maggiore corrente, occorrerà disporre di un driver esterno che possa fornirla. Una soluzione semplice è quella di utilizzare un L293D(www) che è di facile reperibilità:

Si tratta di un integrato in PDIP a 16 pin che non necessita di componenti esterni e può essere cablato facilmente su una breadboard oppure acquistato sotto forma di una delle tante schedine cinesi (Amazon, Banggood, Aliexpress, ecc.).
Contiene 4 semi ponti con i diodi flyback di protezione. Può gestire correnti da 600mA a 1.2A con tensioni fino a Vcc2=36V.
Nel nostro caso sarà opportuno alimentare tutto il circuito con la Vdd a 5V.



In questa esercitazione abbiamo preferito utilizzare un driver basato sull'integrato **MX1508**, di bassissimo costo e facilmente reperibile già installato su una piccola scheda



L'integrato è analogo a L293, essendo costituito da 4 semi ponti, ma è previsto per controllare motori a bassa tensione (non oltre 9V) con una corrente inferiore a 1A.
Dispone di protezione contro la sovra temperatura, con ripristino automatico.
Non sono previsti diodi clamp, ma è sempre opportuno metterli esternamente quando si tratta di comandare motori o carichi induttivi di una certa entità.

La scelta di questa soluzione deriva dalla immediatezza di impiego e dal costo minimo, ma è evidente che si potrà utilizzare un qualsiasi altro driver adeguato, pronto o assemblato come meglio preferite.

Nonostante che questi driver possano essere alimentati con una tensione esterna, per non complicare troppo le cose scegliamo un motore che possa girare con 5V di alimentazione, alimentando motore e scheda di sviluppo dalla stessa fonte..

Se scegliete di utilizzare una Vcc oltre i 5V per alimentare il motore è indispensabile realizzare un cablaggio che eviti la più remota possibilità di ritrovarsi questa tensione sul circuito del microcontroller, cosa che, al minimo, lo distruggerebbe, ma che avrebbe molte probabilità di danneggiare anche il tool di sviluppo. Quindi, **niente cablaggi volanti disordinati ed insicuri.**

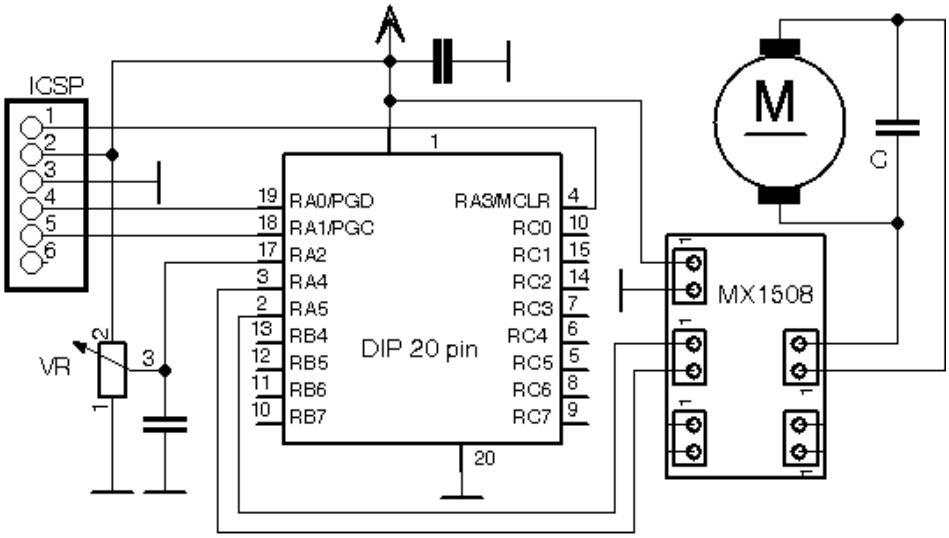

Se state alimentando i circuiti con il PICKIT è necessario inserire una alimentazione esterna perchè la corrente complessivamente assorbita dal circuito supera facilmente quanto disponibile dai pin del tool di sviluppo.

Per quanto riguarda il motore, l'ideale è un consumo a vuoto inferiore a 100mA e una tensione di funzionamento di 5-6V.

E' abbastanza facile trovare in vendita piccoli motori che possano girare a 5V, ma se avete una vecchia unità CD, potete recuperare quello che comanda l'estrazione del cassetto. Sono motori che girano bene anche a 5V, con un consumo a vuoto di meno di 60mA.

Dato che qui ci interessa afferrare il principio dell'applicazione, non ha molto senso usare motori di potenza maggiore, anche perchè sarebbe richiesta una sorgente di alimentazione adatta e un driver esterno adeguato.

Lo schema elettrico:



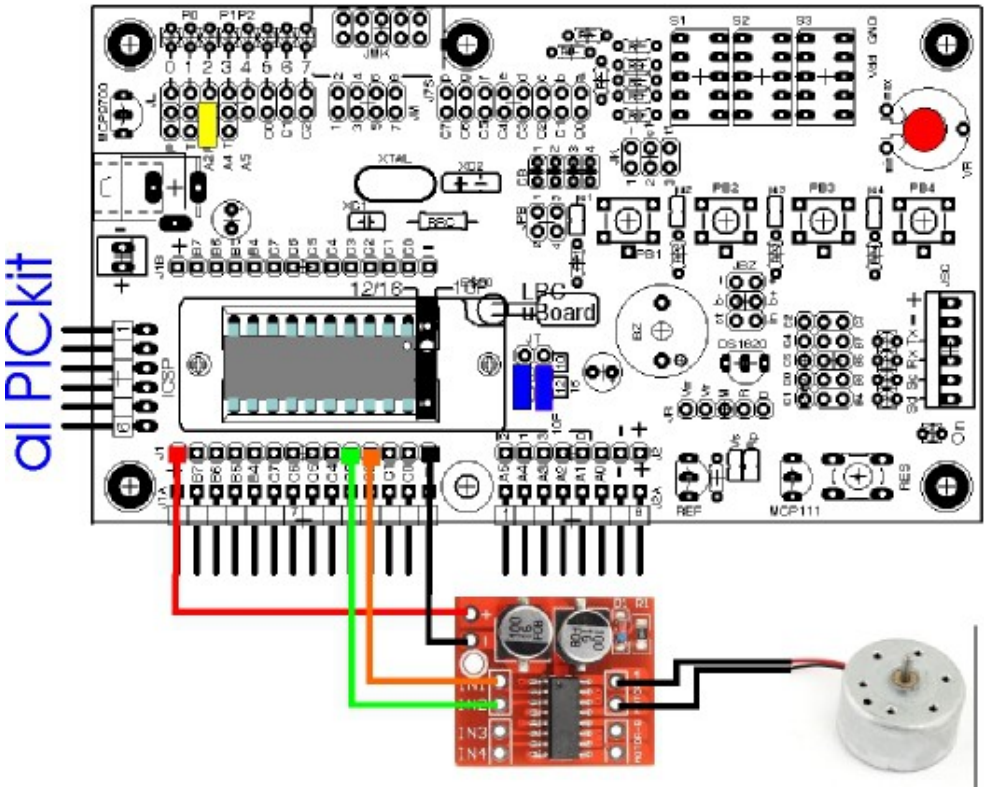
Utilizziamo la configurazione con due mezzi ponti per evitare la necessità di una doppia alimentazione; aggiungiamo un condensatore **C** da 100nF in parallelo al motore per ridurre i picchi di commutazione.

Ci colleghiamo al canale 2 dell'ECCP/PWM in modalità Half-bridge, con i pin rilocati (ovviamente è possibile evitare la rilocazione ed usare i pin di default).
E si potrà comunque utilizzare anche il canale ECCP/PWM1 cambiando i registri per la carica del duty cycle, l'inizializzazione e i pin di uscita.

Colleghiamo il duty cycle alla posizione del cursore del potenziometro, letta dal modulo ADC:

avremo il motore fermo per il cursore a centro e direzione/velocità variabili spostandoci verso gli estremi della corsa.

Sulla scheda di sviluppo:



E' molto opportuno NON FERMARE il rotore con le dita: se applichiamo un carico, o peggio, blocchiamo la rotazione, la corrente assorbita sale da poche decine di mA a molte centinaia e si rischia il surriscaldamento dell'integrato del driver.

Inizializziamo l'I/O, ricordando di non inserire un weak pull-up sul pin di ingresso analogico:

```
WPUA = 0b00111011;              // no wpu su RA2
```

Inoltre, rilochiamo i pin di uscita del PWM su RA4/RA5:

```
// rilocare P2A su RA5  P2B su RA4
    APFCON1bits.CCP2SEL = 1;
    APFCON1bits.P2BSEL = 1;
```

ricordandoci di definirli come uscite:

```
TRISA = 0b11001111;             // RA5:4 out
```

Scegliamo il Timer4. La frequenza del PWM, in questa applicazione, non è libera: occorre verificare, al minimo, per quale valore il motore in condizione di fermo non sia soggetto a oscillazioni.

Non è scopo di questo corso la progettazione di comandi per motori cc, che non è cosa da poco, soprattutto se si deve gestire una certa potenza: in questi casi si dovrà tenere conto anche dei possibili problemi di sovra corrente nel driver, dei tempi di commutazione dei transistor di potenza, dei fenomeni derivati dalla commutazione di carichi induttivi, dalla presenza del collettore, ecc.

Nel nostro caso, non avendo strumentazioni adeguate, si dovrà solamente verificare per quale frequenza del PWM il motore risponde nel modo più stabile, senza vibrazioni o rumori anomali.

Nel nostro esempio, per il motore scelto, una frequenza attorno ai 1000Hz sembra andare meccanicamente bene (se il motore è montato su una "cassa armonica", a rotore fermo, è udibile la frequenza di commutazione). Si potranno provare frequenze da un paio di centinaia a un paio di migliaia di Hz o più.

Calcoliamo i parametri per il timer, ricordando che in questa versione la sola frequenza di alimentazione è **Fosc/4** e sono possibili solo divisori 1:1/1:4/1:16/1:64.

Con ***Fosc = 4MHz***, abbiamo ***Fosc/4 = 1MHz***, ovvero un **periodo di 1us**.

Se scegliamo il **Timer4**: con un **predivisore 1:4**, abbiamo una frequenza di ingresso di **250kHz** (periodo 4us).
Quindi, utilizzando **`PR2=250`** abbiamo un periodo del timer di ***250 * 4 = 1ms***, ovvero 1000Hz esatti.

Il **valore massimo reso dalla conversione AD**, limitato agli 8MSb, è **255**.
Se impostiamo **`PR4=255`** abbiamo in uscita una frequenza di 976.56Hz. Questo va bene per l'applicazione, mentre non viene richiesto un aggiustamento software del valore di uscita della conversione.

```
// setup Timer4 per 980Hz, clock Fosc/4
void initialize_tmr4(void){
    T4CONbits.T4CKPS = 0b01;    // prescaler 1:4 postscaler 1:1
    PR4 = 255-1;
}
```

Inizializziamo il modulo **`ECCP/PWM2`**:

```
 // setup ECCP/PWM2
void initialize_epwm2(void){
    CCPTMRSbits.C2TSEL = 0b01;      // Select Timer4
    CCP2CONbits.P2M = 0b10;         // Half-bridge
    CCP2CONbits.CCP2M =0b1100;      // PxA/PxB active-high
    CCPR2L = 127;                   // duty 50%
    CCP2CONbits.DC2B = 0b00;        // LSB non usati
    //CCP1AS = 0;
    //PWM1CON = 0;
    //PSTR2CON = 0;
}
```

Il convertitore AD viene impostato per **`FRC`**, canale **`AN2`**, risultato **giustificato a sinistra**.

Da notare che nel chip usato (16F1829) il modulo ADC consente di avere su pin esterni sia la Vref+ che la Vref-; poiché questo non ci serve, basta lasciare i bit di controllo relativi al valore di default (0).

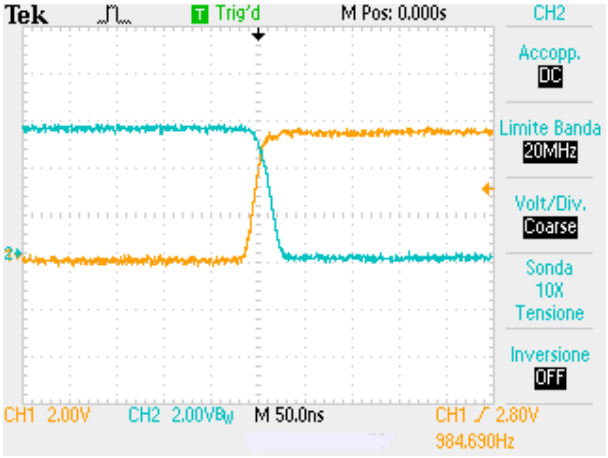
Il main consiste nella sequenza delle inizializzazioni, seguita da un loop che lancia la conversione AD e muove il risultato nel registro del duty cycle.

Ruotando il potenziometro, si avrà una posizione centrale in cui il rotore è fermo; non si tratta di un valore specifico, ma di una piccola area attorno al duty del 50%, dovuta all'inerzia del rotore. Ruotando il cursore al di fuori di questo centro, il rotore girerà in senso orario o antiorario aumentando la velocità, fino al massimo e minimo corrispondenti a duty cycle 0% (cursore a massa) e 100% (cursore alla Vdd).

Se l'associazione della direzione di rotazione rispetto alla posizione del cursore del potenziometro non è quella voluta, si potrà invertire il collegamento del motore con il driver di potenza oppure scambiare **`P2A`** e **`P2B`** all'ingresso della schedina.

Se non avete motore e driver, **potete sostituirli con un LED bicolore rosso/verde a due pin** (con la resistenza in serie), come quello usato in un esempio precedente: ruotando il potenziometro, il colore passerà gradualmente da rosso a verde e viceversa. Nella posizione di fermo saranno accesi in uguale misura entrambi i colori.

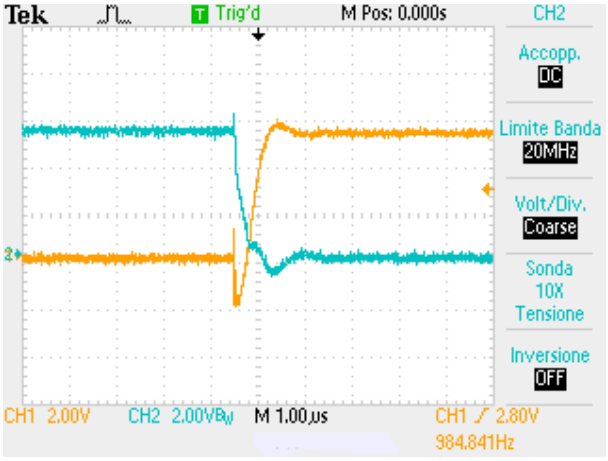
Si potrebbe osservare che, nella descrizione dell'Half-bridge si è parlato di dead band time, ma che qui non abbiamo inserito questo ritardo.



In effetti, se osserviamo la traccia oscilloscopica dei segnali PWM all'uscita del chip, vediamo la rapidità della commutazione, che si sovrappone.

Qua, però, il segnale non scorre nel carico, ma comanda i driver di potenza, quindi non ci sono problemi di corto circuito.

Da osservare che, se volessimo una diversa rapidità di commutazione, non è possibili modificare lo slew rate dei pin, perchè il chip usato non dispone di questa opzione (registri **`SLRCONx`** non presenti) e neppure della possibilità di open drain (registri **`ODCONx`** non presenti).



Diversa è la situazione all'uscita del driver, ai capi del motore.

La commutazione ai capi del motore è tale da non presentare l'inconveniente della conduzione contemporanea dei lati del ponte.

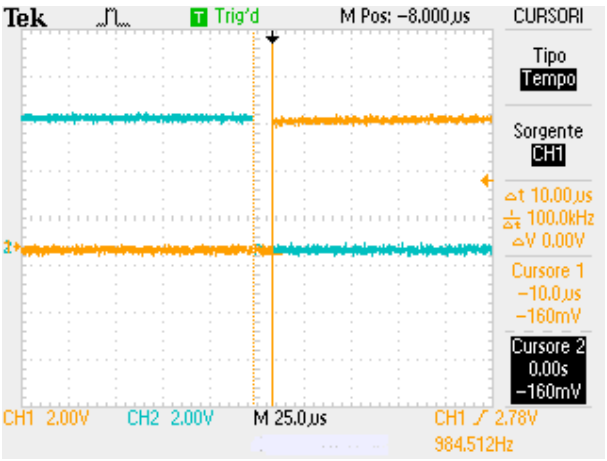
Le caratteristiche del driver, del motore e l'aggiunta del condensatore in parallelo fanno si che i due driver non siano in conduzione contemporanea.

Però, questo è un caso particolare, con basse tensioni e basse correnti. Un reale controllo industriale o di un elettrodomestico avrà una diversa situazione.

Possiamo provare ad inserire una dead band, con il registro **PWM2CON**.

```
PWM2CON = 10;          // deadband 10Tosc = 10us
```

La riga è già presente nel testo sorgente, ma come commento.
Basta de-commentarla per inserire un tempo di ritardo di ***10 * Tosc = 10us***.
Il massimo ottenibile è ***127Tosc***



Nel funzionamento non si notano variazioni: occorre un oscilloscopio per verificare l'effetto dell'istruzione, dove la dead band è ben visibile.

Nell'immagine, i cursori misurano il tempo del ritardo, che è appunto **10us**.

L'inserzione della dead band non può essere casuale (durante questo tempo non viene fornita energia al motore e questo potrebbe essere un problema per l'applicazione).
La determinazione della corretta durata dipende dall'applicazione, da come sono realizzati i driver di potenza, dalle caratteristiche del motore, ecc.

es16C_3

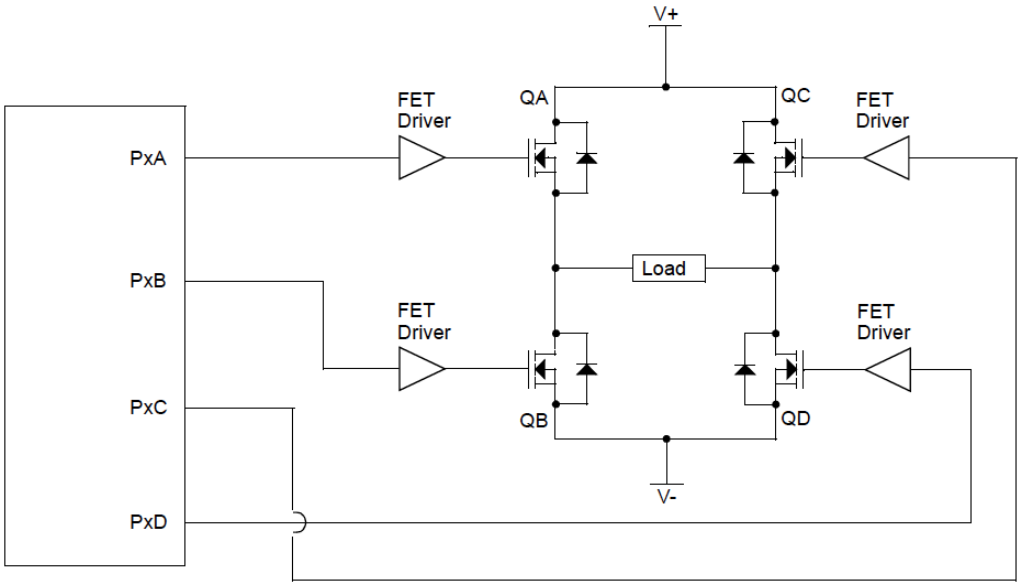
```
/*****************************************************************
*----------------------------------------------------------------
*    Titolo        :  C Enhanced - es16C_3
*                     Half-bridge per motore cc.
*                     Variazione velocità/direzione da potenziometro.
*                     Modulo ECCP/PWM2.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  16F1829
*    Autore        :  afg
*
*----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       16F1829 @ 20 pin
*
*                    __  __
*                   |  \/  |
*              Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*              RC5 -|5   16|- RCO
*              RC4 -|6   15|- RC1
*              RC3 -|7   14|- RC2
*              RC6 -|8   13|- RB4
*              RC7 -|9   12|- RB5
*              RB7 -|10  11|- RB6
*                   |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: Out P2B
*    RA4      3: Out P2B
*    RA3      4: MCLR
*    RC5      5: In
*    RC4      6: In
*    RC3      7: In
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In
*    RC2     14: In
*    RC1     15: In
*    RC0     16: In
*    RA2     17: In  AN2
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/
```

```
// intosc, no clockout, no wdt, no pwrt, mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1829.h"

#include <xc.h>

#define _XTAL_FREQ   4000000        // OSCINT 4MHz

//    FUNZIONI.
// clock a 4MHz
void oscint_4M(void){
    OSCCONbits.IRCF = 0b1101;       // intosc 4MHz
}

// inizializza I/O
void initialize_io(void){
    ANSELA = 0b00000100;            // solo AN2-RA2
    ANSELB = 0;                     // no analogiche
    ANSELC = 0;
    TRISA = 0b11001111;             // RA5:4 out
    WPUA = 0b00111011;              // no wpu su RA2
 // abilita wpu per ingressi
    nWPUEN = 0;
 // rilocare P2A su RA5  P2B su RA4
    APFCON1bits.CCP2SEL = 1;
    APFCON1bits.P2BSEL = 1;
}

// setup Timer4 per 980Hz, clock Fosc/4
void initialize_tmr4(void){
    T4CONbits.T4CKPS = 0b01;    // prescaler 1:4 postscaler 1:1
      PR4 = 255-1;
}
// Timer4 on & clear flag
void tmr4_on(void){
    TMR4IF = 0;
    TMR4ON = 1;
}

 // setup ECCP/PWM2
void initialize_epwm2(void){
    CCPTMRSbits.C2TSEL = 0b01;      // Select Timer4
    CCP2CONbits.P2M = 0b10;         // Half-bridge
    CCP2CONbits.CCP2M =0b1100;      // PxA/PxB active-high
    CCPR2L = 125;                   // duty 50%
    CCP2CONbits.DC2B = 0b00;        // LSB non usati
    //CCP1AS = 0;
    //PWM1CON = 0;
    //PSTR2CON = 0;
    //PWM2CON = 10;                  // dead band 10us
}

/* initializza ADC
   clock FRC, risultato giustificato a sinistra, Vref+ a Vdd
   Vref- a Vss, ingresso da AN2/RA2, no autotrigger */
void initialize_adc(void){
    ADCON1 = 0x70;              // FRC, just left, Vref=Vdd
    ADCON0bits.CHS = 0b00010;   // canale AN2
    ADON = 1;                   // ADC on
```

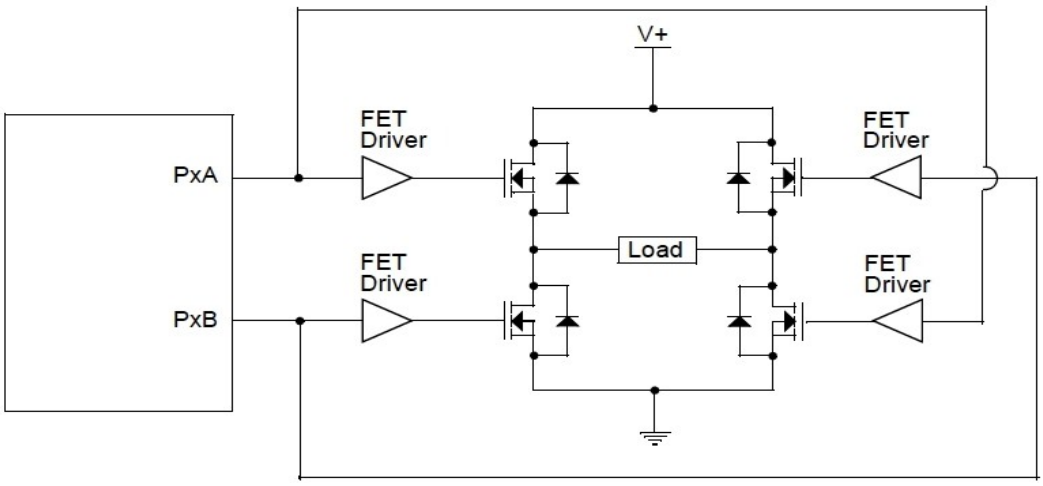
```
}

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

   oscint_4M();
   initialize_io();            // Inizializza I/O
   initialize_tmr4();          // setup Timer
   initialize_epwm2();         // setup ECCP
   initialize_adc();           // setup ADC
   tmr4_on();                  // timer on

// Loop principale
  while (1){

  // conversione AD
   ADCON0bits.GO = 1;                // avvia conversione
     while (ADCON0bits.GO_nDONE) ;   // attesa fine conversione

  // aggiorna regidtro duty cycle
        CCPR2L = ADRESH;            // duty = risultato ADC
    }
}
```

16.4 – Full-bridge per comandare un motore.

Un controllo completo della rotazione del motore si ha utilizzando un Full-bridge. Questo è costituito da 4 driver di potenza, comandati ognuno da un segnale del microcontroller.



Viene utilizzato anche il nome H-bridge, per la forma ad H dei transistor e del carico, ma non va confuso con un doppio Half-bridge visto prima, che pure assume lo stesso nome, ma è comandato da soli due segnali:



Il Full-bridge che utilizziamo qui non va confuso anche con soluzioni integrate in cui è presente una logica di controllo interna, come i vari L298 e simili.
Nel nostro esempio è il modulo **ECCP/PWM** del microcontroller a gestire la logica di controllo; i componenti esterni costituiscono solo un buffer per ottenere le correnti necessarie.

Non c'è molta scelta di oggetti già pronti, come quello utilizzato nella dimostrazione dell'Half-bridge. Si trovano vari controller che forniscono una “intelligenza” locale (L298, DRV8848, TLExxx, MC3391, LMD18200, ecc), ma questi non sono previsti strettamente per essere utilizzati

con EPWM; in genere riducono il numero dei segnali necessari a un PWM, un enable e un segnale di direzione, mentre a volte integrano monitor per il controllo della corrente o protezioni termiche.

Ne consegue che, volendo sperimentare il comando di un Full-bridge gestito dal modulo EPWM, la soluzione praticabile è quella di assemblare i driver di potenza.
La progettazione e la realizzazione di un Full-bridge di potenza non è una cosa da poco, ma se restiamo nell'ambito di basse potenze con la stessa alimentazione del microcontroller, possiamo ricorrere a schemi semplici.

Possiamo utilizzare transistor BJT o MOSFET. Qui trovate schemi e progetti (www) per diversi tipi di Full-bridge con BJT complementari, MOSFET complementari e Darlington.
Per motori di piccola potenza come quello indicato in precedenza, consigliamo il tipo con BJT, che è semplice ed economico e si può realizzare sia su pcb che su breadboard o millefori.
Se avete motori che richiedono maggiore corrente potete usare lo schema con i MOSFET. Entrambi prevedono la stessa tensione di alimentazione del microcontroller.

Vediamo alcuni particolari di quest'ultimo.
Il ponte è composto da due coppie di MOSFET complementari : 2 canale N (Q3-Q4) e 2 canale P (Q1 e Q2).
Le resistenze polarizzano alla tensione positiva (R5 e R6) e negativa (R7 e R8) i gate dei MOSFET, mantenendoli bloccati quando manca un segnale di comando, ad esempio se gli ingressi **`P1A/B/C/D`** non sono collegati al microcontroller oppure i pin del microcontroller sono in tri-state. Questa precauzione è indispensabile per evitare conduzione dei transistor non desiderata prima della presa di controllo da parte dei segnali del PWM.

Durante il funzionamento, se il segnale **`P1A`** è a livello basso, il MOSFET **Q1** entra in conduzione, collegando il capo del motore **MA** alla tensione positiva. Applicando un PWM positivo a **`P1D`** il capo **MB** sarà collegato alla massa attraverso **Q4**: il motore gira in funzione della polarità applicata segnale (MA + e MB -). Contemporaneamente **`P1C`** è a livello alto e blocca **Q2** , mentre **`P1B`** a livello basso blocca **Q3**.

L'inversione della direzione di rotazione si ottiene scambiando le coppie di segnali ai gate dei MOSFET. Quando conducono **Q2** e **Q3**, il capo **MB** è collegato al positivo e **MA** alla massa, invertendo la polarità su motore e quindi il senso di rotazione.
Questi segnali ai giusti livelli sono generati dal modulo ECCP/PWM in modalità Full-bridge.

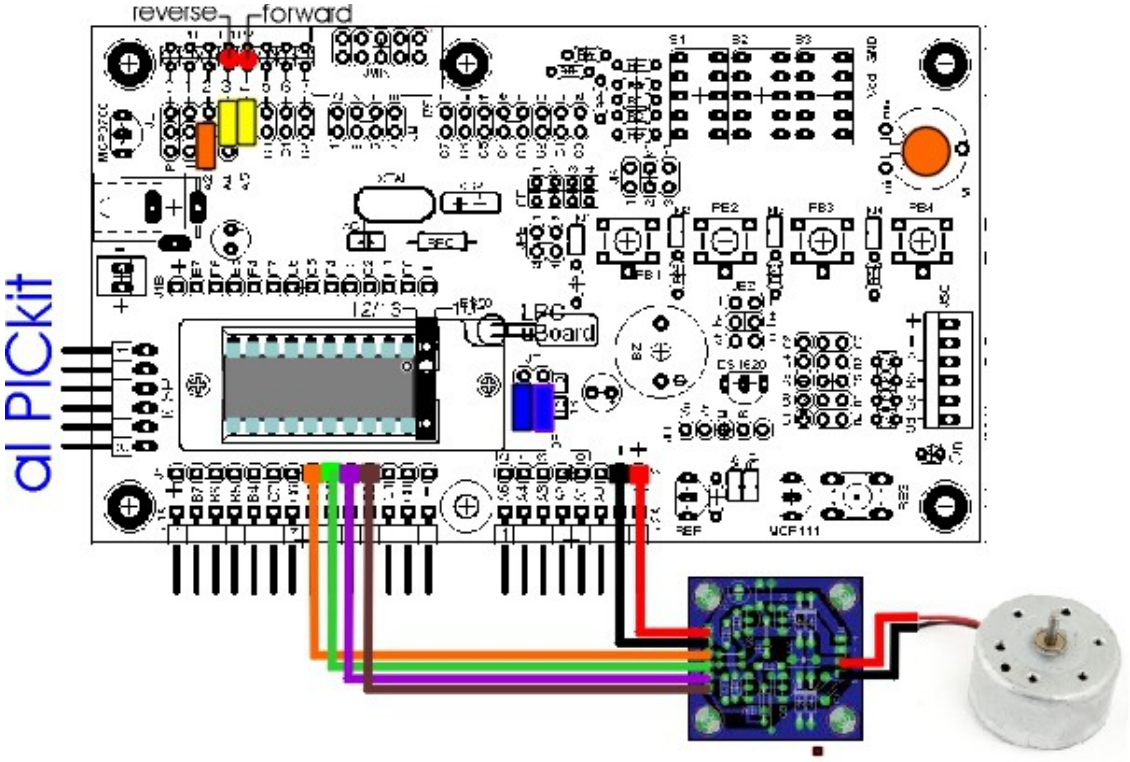
Il microcontroller può comandare i MOSFET P in quanto si tratta di elementi logic gate e la tensione di alimentazione è la stessa Vdd. Se si usasse una tensione maggiore occorrerebbe inserire dei gate driver. Questo non è un problema, ma si complicherebbe inutilmente il circuito per questa esercitazione.

Le resistenze R1/2/3/4 limitano la corrente nel gate.
Quattro diodi proteggono dagli spikes di commutazione con un condensatore in parallelo al motore per ridurre il rumore del collettore. C2 provvede a limitare i disturbi sulla tensione di alimentazione.

I componenti:

Q1, Q2	MOSFET P logic gate	ad es. NDS6020P
Q2, Q3	MOSFET N logic gate	ad es. FQP20N60L
R1,2,3,4	150ohm	
R5,6,7,8	10k	
C1	10-100nF	
C2	10-100uF	
Diodi	Fast recovery o Shottky	ad es. FR104, 1N5822, 1N4448

Sono stati usati MOSFET in contenitore TO-220, del tutto eccessivi per questo esempio, ma si tratta di elementi di facile reperibilità e basso costo. Altre soluzioni circuitali le trovate qui.
Questo ponte può comandare correnti di qualche ampere, ma noi lo useremo con lo stesso motore impiegato nell'esempio precedente, per cui il circuito può essere realizzato su una breadboard, una millefori o su un circuito stampato.



Dobbiamo impiegare **ECCP/PWM1** perchè è il solo modulo a disporre dell'opzione Full-bridge.

Utilizziamo il **Timer2** per il clock e programmiamo la stessa frequenza dell'esempio precedente, cioè circa 1kHz. Come nell'esempio precedente, calcoliamo i parametri per il timer, ricordando che in questa versione la sola frequenza di alimentazione è **Fosc/4** e sono possibili solo divisori 1:1/1:4/1:16/1:64.

Con ***Fosc = 4MHz***, abbiamo ***Fosc/4 = 1MHz***, ovvero un **periodo di 1us**.

Se scegliamo il **Timer4**: con un **predivisore 1:4**, abbiamo una frequenza di ingresso di **250kHz** (periodo 4us). Quindi, utilizzando **`PR2=250`** abbiamo un periodo del timer di ***250 * 4 = 1ms***, ovvero 1000Hz esatti.

Il **valore massimo reso dalla conversione AD**, limitato agli 8MSb, è **255**.
Se impostiamo **`PR4=255`** abbiamo in uscita una frequenza di 980.392Hz. Questo va bene per l'applicazione, mentre non viene richiesto un complicato aggiustamento software del valore di uscita della conversione.

Configuriamo il modulo ECCP1 per la gestione PWM del Full-bridge:

```
/* inizializza ECCP1
   modo selezionato : full-bridge forward
                      P1A active-low
                      P1B active-high
                      P1C active-low
                      P1D active-high */
    CCP1CONbits.P1M = 0b01;      // full-bridge forward
    CCP1CONbits.DC1B = 0b00;      // LSb azzerati
    CCP1CONbits.CCP1M = 0b1110;   // PxA, PxC active-low
                                  // PxB, PxD active-high
```

Va notato che posiamo anche partire con la configurazione "**reverse**":

```
/* inizializza ECCP1
   modo selezionato : full-bridge reverse
                      P1A active-low
                      P1B active-high
                      P1C active-low
                      P1D active-high */
    CCP1CONbits.P1M = 0b11;       // full-bridge reverse
    CCP1CONbits.DC1B = 0b00;      // LSb azzerati
    CCP1CONbits.CCP1M = 0b1110;   // PWM con P1A, P1C active-low
                                  //          P1B, P1D active-high
```

La definizione "**forward**" (avanti) e "**reverse**" (indietro) è relativa esclusivamente alla sequenza dei segnali PWM ed è associabile al senso di rotazione del motore solo in funzione di come questo è collegato al ponte. Anche nel motore il polo "positivo" e "negativo", a parte problemi costruttivi di motori in cui non è prevista l'inversione, è relativo solamente alla convenzione che rende la rotazione oraria o antioraria.
Se prendiamo lo schema del driver e colleghiamo il polo "positivo" del motore al punto MA e il negativo al punto MB, quando entrano in conduzione Q1 e Q4 la polarità applicata al motore è "positiva" ed il rotore girerà in senso orario. Quando vanno in conduzione Q2 e Q3 lq polarità è invertita e si invertirà anche il senso di rotazione.
Quindi, un collegamento tra "forward"/"reverse" e il senso di rotazione dipende da come è collegato il motore rispetto alla sequenza del PWM.

Se osserviamo, i bit **`P1M<1:0>`** del registro **`CCP1CON`** determinano il modo Full-bridge

P1M<1:0>	modo	direzione
00	Single output	-
01	Full-bridge	forward
10	Half-bridge	-
11	Full-bridge	reverse

Quindi, possiamo dire che il bit **P1M0** deve essere a **1** per il Full-bridge, mentre il bit **P1M1** stabilisce la direzione.

Per quanto riguarda il duty cycle, lo colleghiamo al valore reso dalla conversione AD della posizione del cursore del potenziometro.
La conversione a 10bit rende al massimo 1023 per il cursore collegato alla Vdd (usando la Vdd come Vref+). Di questi usiamo gli 8MSb di **ADRESH** (giustificazione a sinistra), scartando i due LSb di **ADRESL** In ogni caso, eliminando questi due bit abbiamo il vantaggio di ridurre i calcoli e di immettere un filtro nel risultato della conversione, che, come abbiamo visto nelle esercitazioni relative al modulo ADC, sono spesso soggetti a instabilità per i rumori nel segnale da convertire.

Il contenuto di **ADRESH** varierà tra 00 e 255.
Combiniamo le cose in modo tale che si abbia una divisione in tre bande:

- **ADRESH** da 0-126 : rotazione in un senso
- **ADRESH** 127 o 128 : fermo
- **ADRESH** 129-255 : rotazione opposta

Per il modulo PWM, con i valori impostati, si ha una variazione del duty cycle da 0 a 100% per una variazione di **CCPR1L:DC1B** da 0 a 1024. In effetti si potrà caricare il registro solo fino a 1023, che corrisponde al 99.9%; però questa imprecisione, pur essendoci, non è rilevabile in questa applicazione.
Se abbiamo 255 in **CCPR1L** e 00 in **DC1B**, il numero complessivo è **11111111 00**, ovvero 1020. Questo riduce ancora il PWM a fondo scala al 99.6%, valore del quale ci accontentiamo, a meno di volerci impegnare in calcoli in virgola mobile che, comunque, dovrebbero essere arrotondati a numeri interi, oppure inserire correzioni all'estremo della gamma, peraltro non significative in questa applicazione.

Quindi, una volta eseguita la conversione, selezioniamo le tre bande in base al risultato della conversione:

```
  if (ADRESH >=128){
      CCPR1L = (ADRESH-128)<<1;
    P1M1 = 0        // Full-bridge forward
}
  else {
    CCPR1L = (127-ADRESH)<<1;
    P1M1 = 1;      // Full-bridge reverse
}
```

Ci si può chiedere il senso del **<<1**.
Ricordiamo che lo shift di una posizione a sinistra corrisponde alla moltiplicazione per 2.
Quindi **`(ADRESH-128)<<1`** equivale a **`(ADRESH-128)*2`**

Perchè, allora, non usare la seconda scrittura, che comunica immediatamente il suo senso?
Semplicemente perchè lo shift viene eseguito con una istruzione in linguaggio macchina, mentre la moltiplicazione richiama le routine matematiche che vanno ad occupare numerosi cicli, aumentando le dimensioni del programma e riducendone la velocità di esecuzione.
In generale, dove è possibile utilizzare shift (moltiplicazioni e divisioni per 2) è sempre il caso di utilizzare questi e non l'operatore aritmetico.

Resta ancora da chiarire perchè moltiplicare per 2.
Se siamo col cursore a massa, abbiamo **`ADRESH = 0`** (ovvero minore di 127), sarà

`CCPR1L = 127-ADRESH = 127`

Se mettessimo questo valore in **`CCPR1L`** avremmo un duty cycle del 50% circa. Noi vogliamo che col cursore in questa posizione si abbia la massima velocità in reverse. Questo si ottiene col massimo duty cycle, per cui basterà moltiplicare per due: 127*2=254.
Sul registro **`CCPR1L:DC1B`** è **`1111111000`** ovvero **`1016`**, che equivale ad un duty cycle del 99.2%.

Se **`ADRESH`** è uguale a 255 (maggiore di 128), sarà

`CCPR1L = ADRESH-128 = 127`

Anche in questo caso basterà moltiplicare per due: 127*2 = 254, col duty cycle del 99.2%
Quindi, la moltiplicazione per 2 è giustificata.

Da quanto sopra si comprende anche che **`ADRESH = 127`** o **`128`** farà si che **`CCPR1L = 0`**, ovvero sospenderà il PWM con un duty cycle dello **0%**.

Da notare che il rotore fermo per l'Half-bridge è comunque una generazione di PWM opposti di pari duty cycle, mentre con il Full-bridge il motore è fermo perchè è completamente sospeso il PWM e quindi non c'è erogazione di corrente.

Possiamo aggiungere due LED la cui accensione indica su un pannello di controllo se il motore gira in un senso (LEDR - reverse) o nell'altro (LEDF - forward) oppure è fermo (entrambi spenti).

Il LED sono spenti se il risultato della conversione AD è 127 o 128. La posizione del cursore corrispondente a questi valori è facile da centrare anche su un potenziometro non di estrema precisione: su una corsa di 270°, il valore di ADRESH varia da 0 a 255, quindi ogni step equivale ad un angoli di 270/255, cioè poco più di 1°. I valori di duty cycle = 0% si trovano, quindi, in 2° attorno alla posizione centrale del cursore.
Per questi valori i LED saranno spenti gli altri due; il duty dello 0% fa si che il rotore sia fermo.

es16C_4

```
/*****************************************************************
*----------------------------------------------------------------
*    Titolo        :  C Enhanced - es16C_4
*                     Full bridge per motore cc.
*                     Variazione velocità/direzione da potenziometro.
*                     Modulo ECCP/PWM1.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  16F1829
*    Autore        :  afg
*
*----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ---------------
*       16F1829 @ 20 pin
*
*                    |‾‾\/‾‾|
*              Vdd -|1   20|- Vss
*       RA5/CLKIN -|2   19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3   18|- RA1/ICSPCLK
*        RA3/MCLR -|4   17|- RA2
*              RC5 -|5   16|- RCO
*              RC4 -|6   15|- RC1
*              RC3 -|7   14|- RC2
*              RC6 -|8   13|- RB4
*              RC7 -|9   12|- RB5
*              RB7 -|10  11|- RB6
*                   |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: Out LEDF
*    RA4      3: Out LEDR
*    RA3      4: MCLR
*    RC5      5: Out P1
*    RC4      6: Out P1
*    RC3      7: Out P1
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In
*    RC2     14: Out P1
*    RC1     15: In
*    RC0     16: In
*    RA2     17: In  AN2
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/
```

```
// intosc, no clockout, no wdt, no pwrt, mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1829.h"

#include <xc.h>

#define _XTAL_FREQ   4000000        // OSCINT 4MHz

#define LEDR    LATAbits.LATA4      // led forward
#define LEDF    LATAbits.LATA5      // led reverse

//    FUNZIONI.
// clock a 4MHz
void oscint_4M(void){
    OSCCONbits.IRCF = 0b1101;       // intosc 4MHz
}

// inizializza I/O
void initialize_io(void){
    ANSELA = 0b00000100;           // solo AN2-RA2
    ANSELB = 0;                    // no analogiche
    ANSELC = 0;
    LATA = 0;                      // preset led spenti
    TRISA = 0b00001111;            // RA5:4 out
    TRISC = 0b11000011;            // RC5:2 out
      WPUA = 0b00111011;             // no wpu su RA2-AN2
 // abilita wpu per ingressi
    nWPUEN = 0;                    // OPTION_REGbits.nWPUEN = 0
}

// setup Timer2 per 980Hz, clock Fosc/4
void initialize_tmr2(void){
    T2CONbits.T2CKPS = 0b01;    // prescaler 1:4 postscaler 1:1
      PR2 = 255-1;
}
// Timer4 on & clear flag
void tmr2_on(void){
    TMR2IF = 0;
    TMR2ON = 1;
}

 // setup ECCP/PWM1
void initialize_epwm1(void){
    //CCPTMRSbits.C1TSEL = 0b00;   // Select Timer2 (defailt)
    CCP1CONbits.P1M = 0b01;        // full bridge forward
      CCP1CONbits.CCP1M =0b1110;     // P1A/C low P1B/D high
    CCP1CONbits.DC1B = 0b00;       // LSB non usati
    //CCP1AS = 0;
    //PWM1CON = 0;
    //PSTR1CON = 0;
}

/* initializza ADC
   clock FRC, risultato giustificato a sinistra, Vref+ a Vdd
   Vref- a Vss, ingresso da AN2/RA2, no autotrigger */
void initialize_adc(void){
    ADCON1 = 0x70;              // FRC, just left, Vref=Vdd
    ADCON0bits.CHS = 0b00010;   // canale AN2
    ADON = 1;                   // ADC on
```

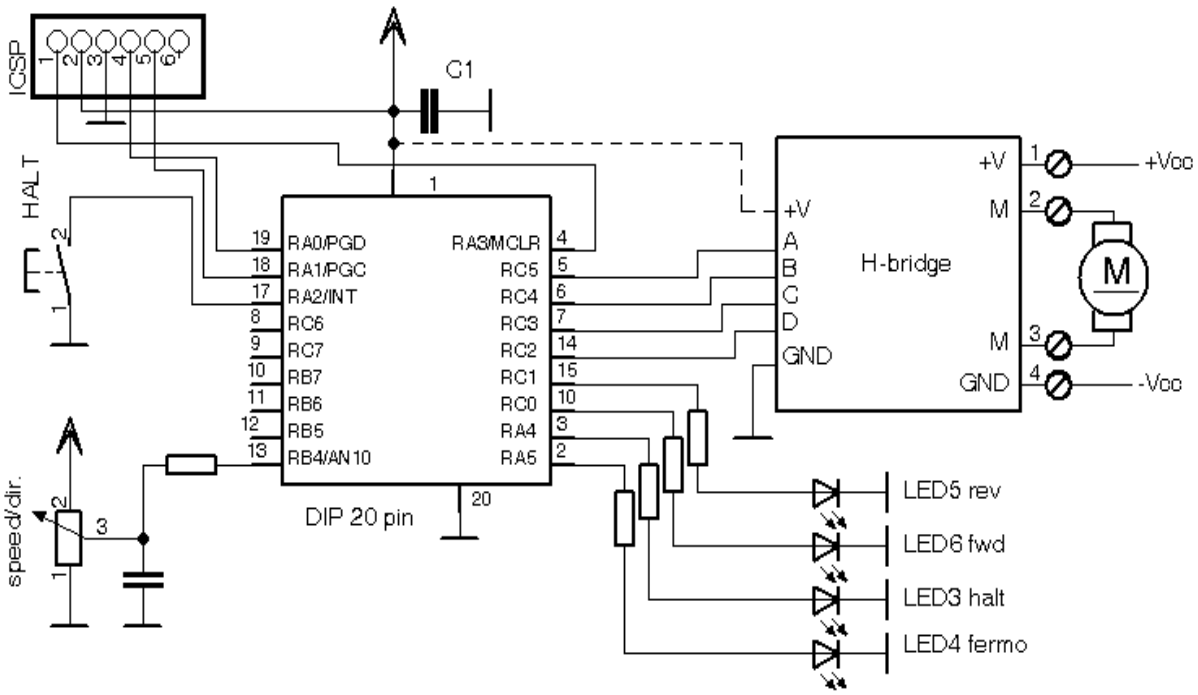
```
}


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

   oscint_4M();
   initialize_io();            // Inizializza I/O
   initialize_tmr2();          // setup Timer
   initialize_epwm1();         // setup ECCP
   initialize_adc();           // setup ADC
   tmr2_on();                  // timer on

// Loop principale
  while (1){

  // conversione AD
   ADCON0bits.GO = 1;                // avvia conversione
   while (ADCON0bits.GO_nDONE) ;     // attesa fine conversione

   if(ADRESH >= 128){
     CCPR1L = ((ADRESH - 128)<<1) ;
     CCP1CONbits.P1M = 0b01;
       if (ADRESH == 128){             // se 128, spegni led
        LEDF=LEDR=0;
     }
       else{
      LEDF = 1;   LEDR =0;
     }
   }
   else {
      CCPR1L = ((127 - ADRESH)<<1) ;
      CCP1CONbits.P1M = 0b11;
       if (ADRESH == 127){             // se 128, spegni led
        LEDF=LEDR=0;
     }
       else{
      LEDF = 0; LEDR=1;
     }
   }
  }
}
```

16.5 – Auto shutdowm e restart.

Con un circuito come quello visto sopra, possiamo aggiungere e sperimentare la funzione di auto shutdown e restart.

Lo scopo di questa opzione è quello di bloccare automaticamente il PWM (e quindi l'invio di energia al carico) in caso di condizioni anomale, ad esempio per una sovracorrente, oppure come stop di emergenza per bloccare la rotazione del motore.
Nel primo caso si potrà monitorare la corrente misurando la caduta di tensione su un resistore in serie al carico con un comparatore o col modulo ADC; nel secondo si dovrà rispondere con rapidità ad un comando esterno.

Se non inseriamo l'auto restart, occorrerà un intervento del programma per ripristinare il funzionamento regolare. Se implementiamo l'auto restart, al cessare della condizione anomala, il PWM riprenderà automaticamente.



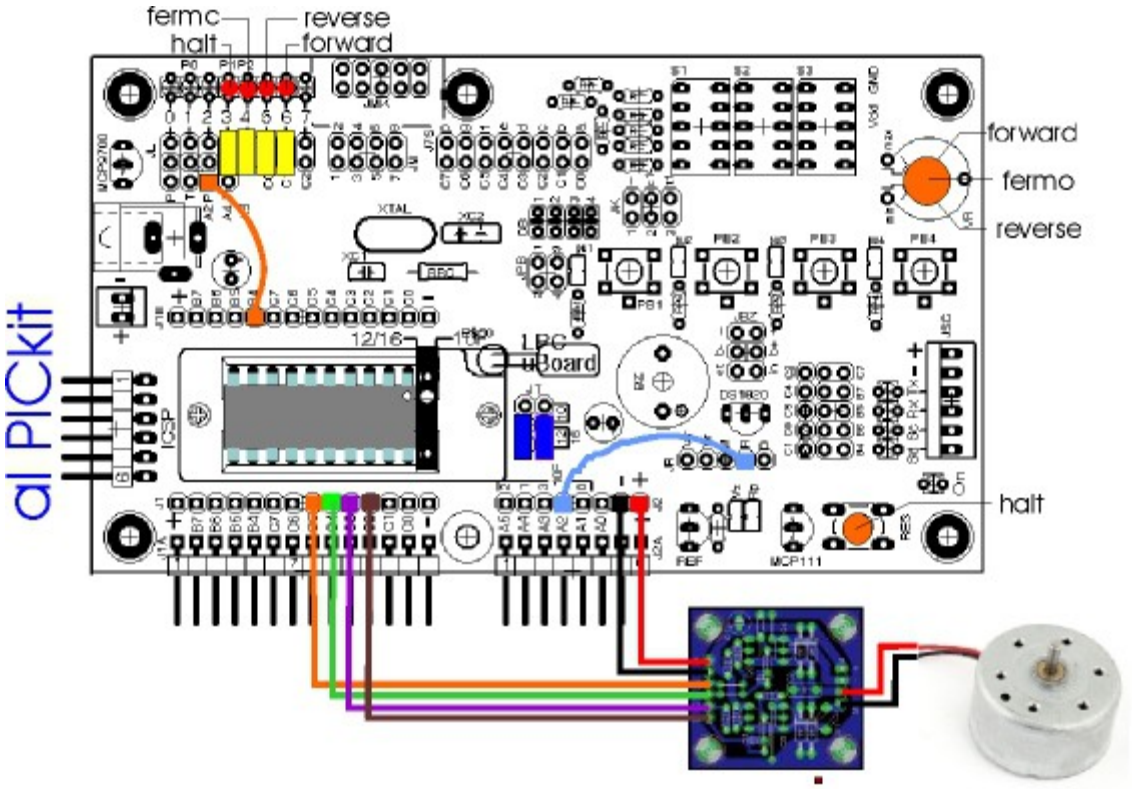
L'ingresso del segnale di arresto esterno è il pin **`INT`** (**`RA2`**), che è una delle opzioni specifiche della funzione di shutdown, garantendo così la massima rapidità di intervento.

Dato che **`INT`** occupa **`RA2`**, spostiamo l'ingresso analogico su **`RB4`** che è **`AN10`**.

Per identificare la situazione abbiamo aggiunto 4 LED:

- un LED indica la direzione avanti (LEDF - forward)
- un LED indica quella indietro (LEDR – reverse)
- un LED indica la posizione di motore fermo (LEDS)
- un LED indica l'intervento dello shutdown (LEDH)

Sulla scheda di sviluppo:



Utilizziamo il pulsante RES per portare a livello 0 il pin **INT**.

Nel programma, basterà variare le attribuzioni degli I/O e impostare il pin **INT** (**RA2**) come ingresso. Non è necessario agire sul bit **INTE** perchè qui non ci occorre aggiungere una interruzione, che potrà essere utile per comunicare al programma l'avvenuto evento di emergenza.

Nella programmazione del modulo ECCP/PWM, aggiungiamo l'attivazione dell'auto shutdown dipendente dallo stato del pin **INT**; da ricordare che per lo shutdown conta lo stato del pin e non la transizione, che, invece, è determinante per la chiamata di una interruzione.

Abbiamo diverse possibilità di configurazione dei pin di uscita del PWM durante lo shutdown, per adattarli all'hardware del ponte.
Per il ponte che stiamo usando, la coppia **P1A/P1C** può essere mandata in tri-state (alta impedenza); questa situazione è pari a quella di avviamento del sistema, ovvero i driver di potenza saranno mantenuti inattivi dai relativi resistori sulla base/gate.

Possiamo anche scegliere di mandare a livello alto le due uscita, confermando con il segnale dal micro il blocco dei driver.

```
CCP1ASbits.PSS1AC = 0b11;       // P1A/P1C tris-state
//CCP1ASbits.PSS1AC = 0b01;      // P1A/P1C = 1 driver off
```

una alternativa viene compilata; l'altra, come commento, potrà essere usata scambiandola con la prima.

Per quanto riguarda **P1B/P1D** la scelta è il tri-state, analogamente alle uscite precedenti, oppure il livello 0 che conferma il blocco dei driver, ma anche la possibilità di averli entrambi attivi: questa soluzione cortocircuita il motore attraverso la massa e opera una frenatura.

A seconda delle caratteristiche del motore, compilando la scelta del tri-state o del freno avremo due diversi effetti: con il tri-state ruoterà ancora un attimo, mentre con il freno si arresterà immediatamente. La visibilità della differenza dipenderà dall'inerzia del rotore.

```
//CCP1ASbits.PSS1BD = 0b11;       // P1B/P1D tri-state
CCP1ASbits.PSS1BD = 0b01;     // P1B/P1D = 1 driver on - freno
//CCp1ASbits.PSS1BD = 0b00;     // P1B/P1D = 0 driver off
```

Per il restart, scegliamo la modalità automatica, ovvero, non appena cessa la causa dell'arresto, il movimento riprende.

```
PWM1CONbits.P1RSEN = 1;         // auto restart
//PWM1CONbits.PRSEN = 0;          // manual restart
```

In questo modo, la rotazione riprende immediatamente, controllata dal duty cycle del momento.

Esiste anche la possibilità di comandare “manualmente” il restart: occorre scrivere le linee di programma che cancellano il bit ASE e riavviano la rotazione; questa scelta potrà essere quella preferibile nel caso in cui si voglia portare a zero il duty cycle prima di far riavviare il motore o altre azioni di sicurezza o dettate dalla struttura di quanto comandato dal motore.

Se portiamo in rotazione (non importa il verso) il motore e premiamo RES, il movimento si arresterà. Rilasciando il pulsante il movimento riprenderà.

Per identificare la situazione abbiamo 4 LED:

- un LED indica la direzione avanti e un LED quella indietro (LEDR – reverse e LEDF - forward)
- un LED indica la posizione di motore fermo (LEDS)
- un LED indica l'intervento dello shutdown (LEDH)

Il LED di stop viene acceso se il risultato della conversione AD è 127 o 128. La posizione del cursore corrispondente a questi valori è facile da centrare anche su un potenziometro non di estrema precisione: su una corsa di 270°, il valore di ADRESH varia da 0 a 255, quindi ogni step equivale ad un angoli di 270/255, cioè poco più di 1°. I valori di duty cycle = 0% si trovano, quindi, in 2° attorno alla posizione centrale del cursore.
Per questi valori il LEDS sarà acceso e spenti gli altri due; il duty dello 0% fa si che il rotore sia fermo.

Il LEDH si accenderà quando è attivo l'auto shutdown.

Se nella condizione di arresto variamo il cursore del potenziometro, varieranno anche duty e/o direzione, anche se, essendo le uscite in shutdown, non ci sarà segnale PWM sui pin.
Per ripartire dalla posizione di fermo, possiamo, ad esempio, ruotare il cursore fino a che si accende il LED si STOP, poi rilasciare il pulsante RES.

es16C_5

```
/*****************************************************************
*-----------------------------------------------------------------
*    Titolo        :  C Enhanced - es16C_5
*                     Full bridge per motore cc.
*                     Variazione velocità/direzione da potenziometro.
*                     Auto shutdown e restart da INT.
*                     Modulo ECCP/PWM1.
*    Data          :  01-05-2011
*    Modificato il :
*    Versione      :  V0.0
*    Ref. Hardware :  16F1829
*    Autore        :  afg
*
*-----------------------------------------------------------------
*****************************************************************
*  Impiego pin :
*  ----------------
*       16F1829 @ 20 pin
*
*                    __  __
*                   |  \/  |
*              Vdd -|1    20|- Vss
*       RA5/CLKIN -|2    19|- RA0/ICSPDAT
*      RA4/CLKOUT -|3    18|- RA1/ICSPCLK
*        RA3/MCLR -|4    17|- RA2
*              RC5 -|5    16|- RCO
*              RC4 -|6    15|- RC1
*              RC3 -|7    14|- RC2
*              RC6 -|8    13|- RB4
*              RC7 -|9    12|- RB5
*              RB7 -|10   11|- RB6
*                   |______|
*
* Impiego pin:
*    Vdd      1: ++
*    RA5      2: Out LEDStop
*    RA4      3: Out LEDHalt
*    RA3      4: MCLR
*    RC5      5: Out P1A
*    RC4      6: Out P1B
*    RC3      7: Out P1C
*    RC6      8: In
*    RC7      9: In
*    RB7     10: In
*    RB6     11: In
*    RB5     12: In
*    RB4     13: In  AN10
*    RC2     14: Out P1D
*    RC1     15: Out LEDReverse
*    RC0     16: Out LEDForward
*    RA2     17: In  INT
*    RA1     18: In
*    RA0     19: In
*    Vss     20: --
*
* Note: - INTOSC default @ 500kHz*
*****************************************************************/
```

```
// intosc, no clockout, no wdt, no pwrt, mclr, bor, no code prot.
// no WRT, no pll, no stack error, bor low, no bor lp, no LVP
#include "C:/Corso_C/MyIncludes/conf1829.h"

#include <xc.h>

#define _XTAL_FREQ   4000000        // OSCINT 4MHz

#define LEDR    LATCbits.LATC0      // led forward
#define LEDF    LATCbits.LATC1      // led reverse
#define LEDS    LATAbits.LATA5      // led stop - rotore fermo
#define LEDH    LATAbits.LATA4      // led halt - shutdown

//    FUNZIONI.
// clock a 4MHz
void oscint_4M(void){
    OSCCONbits.IRCF = 0b1101;       // intosc 4MHz
}

// inizializza I/O
void initialize_io(void){
    ANSELA = 0;                     // no analogiche
    ANSELB = 0b00010000;            // AN10 - RB4
    ANSELC = 0;
    LATA = 0;                       // preset led spenti
    LATC = 0;
    TRISA = 0b00001111;             // RA5:4 out
    TRISC = 0;                      // RC  out
    WPUB = 0b00101111;              // no wpu su RB4-AN10
 // abilita wpu per ingressi
    nWPUEN = 0;                     // OPTION_REGbits.nWPUEN = 0
}

// setup Timer2 per 980Hz, clock Fosc/4
void initialize_tmr2(void){
    T2CONbits.T2CKPS = 0b01;        // prescaler 1:4 postscaler 1:1
      PR2 = 255-1;
}
// Timer4 on & clear flag
void tmr2_on(void){
    TMR2IF = 0;
    TMR2ON = 1;
}

 // setup ECCP/PWM1
void initialize_epwm1(void){
    //CCPTMRSbits.C1TSEL = 0b00;    // Select Timer2 (defailt)
    CCP1CONbits.P1M = 0b01;         // full bridge forward
    CCP1CONbits.CCP1M =0b1110;      // P1A/C low P1B/D high
    CCP1CONbits.DC1B = 0b00;        // LSB non usati
    CCP1ASbits.CCP1AS = 0b100;      // vil sul pin INT
    CCP1ASbits.PSS1AC = 0b11;       // P1A/P1C tris-state
    //CCP1ASbits.PSS1AC = 0b01;      // P1A/P1C = 1 driver off
    //CCP1ASbits.PSS1BD = 0b11;      // P1B/P1D tri-state
    CCP1ASbits.PSS1BD = 0b01;       // P1B/P1D = 1 driver on - freno
    //CCp1ASbits.PSS1BD = 0b00;      // P1B/P1D = 0 driver off
    PWM1CONbits.P1RSEN = 1;         // auto restart
    //PWM1CONbits.PRSEN = 0;         // manual restart
```

```
    //PSTR1CON = 0;
}

/* initializza ADC
   clock FRC, giustificato a sinistra, Vref+ a Vdd
   Vref- a Vss, ingresso da AN10/RB4, no autotrigger */
void initialize_adc(void){
    ADCON1 = 0x70;              // FRC, just left, Vref=Vdd
    ADCON0bits.CHS = 0b01010;   // canale AN10
    ADON = 1;                   // ADC on
}


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

   oscint_4M();
   initialize_io();            // Inizializza I/O
   initialize_tmr2();          // setup Timer
   initialize_epwm1();         // setup ECCP
   initialize_adc();           // setup ADC
   tmr2_on();                  // timer on

// Loop principale
  while (1){

// test per shutdown
   if(CCP1ASE){                        // se shutdown
     LEDH = 1;                         // LEDH acceso
   }
   else{                               // altrimenti
     LEDH = 0;                         // LEDH spento
   }

// conversione AD
   ADCON0bits.GO = 1;                  // avvia conversione
   while (ADCON0bits.GO_nDONE) ;       // attesa fine conversione

// test per direzione
   if(ADRESH >= 128){                  // se forward
     CCPR1L = ((ADRESH - 128)<<1);     // carica duty
     CCP1CONbits.P1M = 0b01;           // direzione forward
      if (ADRESH == 128){              // se 128
        LEDF=0; LEDS =1;               // acceso solo LEDS
      }
      else{                            // altrimenti
        LEDS= 0; LEDF = 1;             // acceso LEDF
      }
   }
   else {                              // se reverse
     CCPR1L = ((127 - ADRESH)<<1);     // carica duty
     CCP1CONbits.P1M = 0b11;           // direzione reverse
      if(ADRESH == 127){               // se 127
        LEDR=0; LEDS =1;               // acceso solo LEDS
      }
      else{                            // altrimenti
        LEDS= 0; LEDR=1;               // acceso LEDR
      }
   }
```

```
    }
}
```

Moduli ECCP/PWM.

I moduli **ECCP/PWM** possono svolgere le funzioni dei moduli **CCP/PWM**, con l'aggiunta di alcune opzioni avanzate (**EPWM**), che permettono:

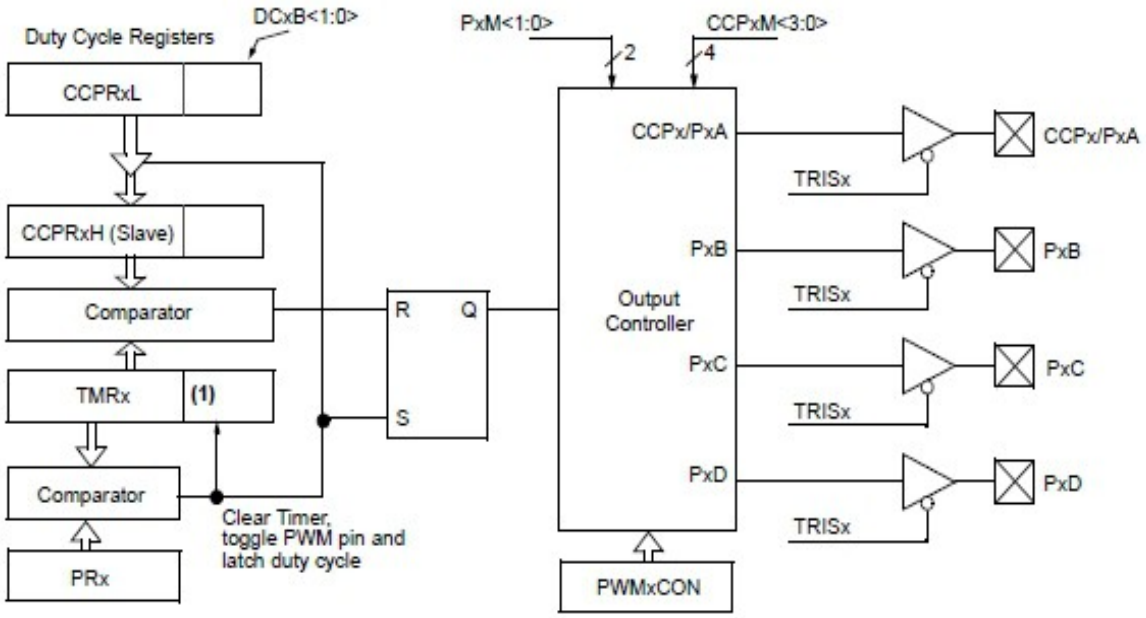
- **PWM singolo**
- **Half-bridge PWM**
- **Full-Bridge PWM, Forward mode**
- **Full-Bridge PWM, Reverse mode**
- **PWM steering**

L'orientamento è quello verso il controllo di carichi di potenza, come motori, SMPS, inverter, ecc. Se non si fa uso di queste modalità enhanced, il modulo ECCP/PWM può essere impiegato analogamente ad un modulo CCP/PWM, con piccole differenze nei registri.

I moduli **ECCP/PWM** possono essere presenti assieme ad altri moduli PWM, oppure anche singolarmente in chip con pochi pin; ci sono ulteriori due tipologie:

- con PWM in grado di gestire solo Half-bridge (due sole uscite)
- oppure con PWM per Full-bridge (4 uscite).

La struttura generale del modulo è la seguente:



Se la struttura di fondo è simile a quelle già viste per gli altri moduli PWM, abbiamo alcune significative differenze:

1. la prima, più evidente, consiste nella disponibilità di **4 pin di uscita**, (**`PxA`**, **`PxB`**, **`PxC`**, **`PxD`**) gestiti da un ***Output Controller***
2. il registro del duty cycle a 10bit è suddiviso in un **`CCPRxL`** che contiene gli 8 MSb, mentre i

due LSb non dispongono di un proprio registro, ma sono i bit **DCxB<1:0>** del registro **CCPxCON** (bit 5:4).

La presenza di più uscite è determinata dalla necessità di comandare driver esterni di potenza a ponte o mezzo ponte, mentre è possibile utilizzare uscite in modo steering o singolo (similmente a CCP).

1. **Tutte le uscite dipendono da un unico timer , quindi la frequenza del PWM sarà la stessa per tutte.**
2. **I latch di queste uscite sono nell'*Output Controller* e non dipendono dai latch dei port.**
3. Il collegamento tra l'uscita PWM e il pin corrispondente dipende, invece, dai **TRIS** del port: **per ottenere l'uscita occorre portare a 0 il bit relativo nel registro di direzione.**

In pratica, il **TRIS** può essere sfruttato come abilitazione dell'uscita, commutando la direzione in ingresso (**TRISx=1**), il che porta il pin in uno stato di alta impedenza. In questo caso sarà necessario polarizzare esternamente gli ingressi dei driver di potenza con pull-up o pull-down a seconda dei casi.

Le uscite dipendono dal modo di funzionamento scelto:

Modo ECCP	PxM<1:0>	PxA	PxB	PxC	PxD
Singolo	00	steering	steering	steering	steering
Half-bridge	10	modulated	modulated	-	-
Full-bridge	01	active	inactive	inactive	modulated
FB reverse	11	inactive	modulated	active	inactive

- ***Active*** indica che il pin è comandato per una uscita al livello definito dai bit **CCPxM<3:0>** del registro **CCPxCON**. Questo segnale manda in conduzione il driver a cui è collegato.
- ***Inactive*** indica che il driver a cui è collegato questo segnale non deve essere in conduzione.
- ***Modulated*** indica un pin a cui è applicato il segnale PWM.

Questo vuol dire che le polarità ***active*** e ***inactive*** devono trovare riscontro nell'hardware del ponte. Dato che è possibile disporre di diverse combinazioni circuitali per i driver di potenza, solitamente è possibile adattare il comando praticamente a tutte le tipologie. Dove questo non è possibile, occorrerà utilizzare driver invertenti. Così pure per la polarità dell'impulso del PWM, anche se, in generale, l'impulso positivo è quello comunemente usato.

In genere, un Full-bridge è realizzato con due coppie complementari di MOSFET o BJT oppure quattro MOSFET della stessa polarità, dove sarà indispensabile l'uso di driver (che sono sempre necessari quando la tensione del ponte è maggiore di quella di alimentazione del microcontroller).

Il foglio dati presenta vari diagrammi dei segnali nelle diverse modalità.
La definizione del PWM richiede la scrittura del registro **CCPxCON**:

bit	7	6	5	4	3	2	1	0
CCPxCON	**PxM<1:0>**		**DCxB<1:0>**		**CCPxM<3:0>**			

```
bit7-6  PxM<1:0>  configurazione output (per modo PWM)
          00   singola uscita PxA; PxB/C/D come GPIO
          01   Full-bridge: PxD modulato, PxA attivo, altri inattivi
          10   Half-bridge: PXA/D modulati con dead band, altri GPIO
          11   Full-bridge invertito: PxB modulato, PxC attivo; altri inattivi

bit5-4  DCx1B<1:0>   LSb del duty cycle. Altri 8MSb in CCPR1L

bit3-0  CCPxM<3:0>   modo (per ECCP)
          0000   reset
          1100 = PWM mode: PxA, PxC active-high; PxB, PxD active-high
          1101 = PWM mode: PxA, PxC active-high; PxB, PxD active-low
          1110 = PWM mode: PxA, PxC active-low;  PxB, PxD active-high
          1111 = PWM mode: PxA, PxC active-low;  PxB, PxD active-low
         le altre combinazioni sono riservate ai modi Compare e Capture
```

Raccogliendo in una tabella le combinazioni dei bit che determinano la polarità dei pin:

CCPxM<3:0>	PxA	PxB	PxC	PxD
1100	*active high*	*active high*	*active high*	*active high*
1101	*active high*	*active low*	*active high*	*active low*
1110	*active low*	*active high*	*active low*	*active high*
1111	*active low*	*active low*	*active low*	*active low*

Se sono disponibili solo due pin di uscita, **PxA** e **PxB**, perchè il modulo non dispone dell'opzione Full-bridge, la tabella vale per comunque per questi due pin.

active high o ***active low*** si riferiscono al livello del pin quando è ***active*** come definito dai bit **PxM<1:0>** , ma anche al livello dell'impulso dei pin ***modulated***.

Da notare che i pin sono definiti a coppie: **PxA** e **PxC** hanno la stessa polarità, così pure **PxB** e **PxD.** Se, per la tipologia del ponte che si vuole usare, nessuna delle combinazioni è adeguata, si dovranno invertire i segnali necessari con driver invertenti.

Le polarità di uscita PWM devono essere selezionate prima di abilitare i driver di uscita pin PWM. La modifica della configurazione della polarità mentre i driver di uscita pin PWM sono abilitati non è consigliata poiché potrebbe causare danni ai circuiti dell'applicazione.

Cancellare il registro **CCPxCON** disabilita il modulo e rende inattive le uscite PWM al loro livello di default: il loro controllo non dipenderà più dall'*Output Controller*, ma dal latch del PORT relativo e dalla direzione dal TRIS e si potranno impostare le funzioni alternative dei pin.

I fogli dati indicano alcune precauzioni da considerare.
I latch dei pin di uscita **PxA**, **PxB**, **PxC** e **PxD** potrebbero non essere negli stati corretti quando il modulo EPWM è disattivato. L'abilitazione dei driver di uscita dei pin PWM prima della definizione delle modalità EPWM può causare danni al circuito dell'applicazione

Per la massima sicurezza, le modalità EPWM devono essere abilitate e va completato un ciclo PWM completo prima di abilitare i driver di uscita pin PWM.

Il completamento di un ciclo PWM può essere determinato monitorando il bit di overflow per il timer selezionato per controllare il modulo. Questo bit di overflow del timer (**`TMRxIF`**) andrà a 1 all'inizio del secondo periodo del segnale PWM.

Sostanzialmente il concetto della sicurezza è il seguente:

- se dal modulo PWM dipende un hardware che potrebbe avere danni da una applicazione casuale dei segnali PWM, occorre che la sezione dell'elettronica di potenza disponga di **pull-up o pull-dowm che polarizzino gli ingressi dei segnali in modo da non assumere configurazioni dannose all'accensione**, quando i pin del microcontrolle sono in tri-state o se il microcontroller non è collegato.
- In secondo luogo, occorrerà che il **modulo PWM sia completamente configurato e attivo correttamente** prima di applicare i segnali ai driver (per questo si indica un ciclo a vuoto); solo dopo questo si potranno abilitare le uscite dei segnali.

I modi PWM determinano la situazione delle uscite, secondo la seguente tabella:

PxM1:0	Funzione
00	**Uscita singola**: **`CCPx/PxA`** modulato, altri con le funzioni alternative o steering sugli altri pin
01	**Full-bridge forward**: **`PxD`** modulato; **`PxA`** attivo; **`PxB, PxC`** inattivi
10	**Half-bridge**: **`PxA/PxB`** modulati con dead band. **`PxC/PxD`** altre funzioni
11	**Full-bridge reverse**. **`PxB`** modulato, **`PxC`** attivo, **`PxA/PxD`** inattivi

Possiamo derivare che

- il bit **`PxM0 = 1`** abilita il modo Full-bridge, mentre
- il bit **`PxM1`** indica la direzione forward-reverse.

Se il bit **PxM0 =0** abbiamo il modo singolo o l'Half-bridge, a seconda del valore di **PxM1**.

La situazione dei bit di **`CCP1CON`** e le funzioni dei pin per il PWM può essere riassunta, quindi, nella seguente tabella:

Modo	CCPxCON	pin	pin	pin	pin
Singolo PWM	**`00xx11xx`**	**`CCPx/PxA`**	`gpio`	`gpio`	`gpio`
Steering		**`PxA`**	**`PxB`**	**`PxC`**	**`PxD`**
Half-bridge	**`10xx11xx`**	**`PxA`**	**`PxB`**	`gpio`	`gpio`
Full-bridge	**`x1xx11xx`**	**`PxA`**	**`PxB`**	**`PxC`**	**`PxD`**

Per i moduli ECCP/PWM con capacità di comando del solo Half-bridge, in pin disponibili sono solo **`PxA`** e **`PxB`**, sia per Half-bridge che per Steering.

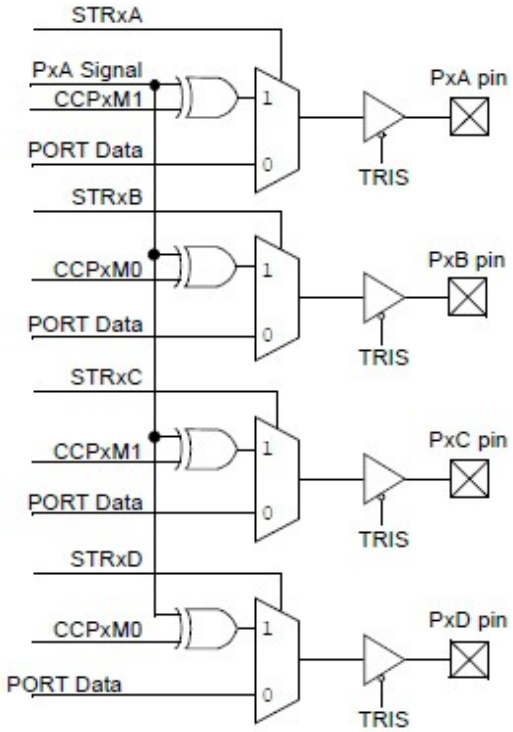
Ricordiamo ancora che, per avere le uscite PWM, occorre che:

- **i pin utilizzati devono essere assegnati con PPS (o eventualmente ri-allocati con APFCON, a seconda del chip in uso)**
- **le funzioni analogiche sui pin interessati devono essere disabilitate.**
- **i pin usati per il PWM devono essere impostati come uscite nel relativo TRIS.**

PWM singolo e Steering.

Nel modo singolo, solo un pin è l'uscita del PWM, mentre le altre tre sono I/O generici. Nella modalità **Steering**, possiamo attivare più uscite per lo stesso segnale PWM, con polarità programmabili.

Nella modalità **Steering**, possiamo attivare più uscite per lo stesso segnale PWM, con polarità programmabili.



Il modo può essere programmato partendo dalla modalità Singola e consente a **qualsiasi delle uscite PWM di essere modulata**.

Inoltre, lo **stesso segnale PWM può essere disponibile contemporaneamente su più pin**.

Una volta selezionata la modalità **Single Output** (**`CCPxM<3:2> = 11`** e **`PxM<1:0> = 00`** del **`CCPxCON`**), il programma potrà può ottenere l'uscita dello stesso segnale PWM su uno, due, tre o quattro pin. Mentre il modo PWM Steering è attivo, **`CCPxM<1:0>`** del registro **`CCPxCON`** selezionano la polarità per i pin **`Px<D:A>`.**

Come solito, i pin di uscita dipendono dal relativo TRIS.

L'opzione di auto spegnimento (auto shutdown), descritta più avanti, si applica anche al modo Steering.

Da osservare che **il segnale PWM è uguale per tutte le uscite abilitate**: non si tratta di quattro PWM separati, ma di espandere lo stesso segnale su più uscite.
I pin possono essere abilitati come uscite del PWM o rilasciati alle loro funzioni generali con

CCPxCON, ma si possono disabilitare le le uscite anche con i relativi **TRIS**.

Va sempre considerato che, a seconda dei chip, sarà possibile ri allocare la funzione di uscita PWM con **APFCON** oppure sarà necessario assegnarla con **PPS.**

Qui dobbiamo selezionare il modo singolo con il registro **CCPxCON**:, si per disporre di un segnale che per aggiungere anche gli altri tre.

Modo	CCPxCON	pin	pin	pin	pin
Singolo PWM	**00xx11xx**	**CCPx/PxA**	gpio	gpio	gpio

L'uscita singola sarò sul pin **CCPx/PxA**.
In steering il modo è lo stesso:

Modo	CCPxCON	pin	pin	pin	pin
Steering	**00xx11xx**	**CCPx/PxA**	**PxB**	**PxC**	**PxD**

Per i moduli ECCP/PWM con capacità di comando del solo Half-bridge, in pin disponibili sono solo **PxA** e **PxB**, sia per Half-bridge che per Steering.

Il controllo dei pin di steering è demandato al registro **PSTRxCON**:

bit	7	6	5	4	3	2	1	0
PSTRxCON	**-**	**-**	**-**	**STRxSYNC**	**STRxD**	**STRCxC**	**STRxB**	**STRxA**

```
bit7-5     non implementati
bit4       STRxSYNC  sincronismo dello steering
             0 = uscita steering aggiornata nel prossimo periodo PWM
             1 = uscita steering aggiornata n ella prossima istruzione
bit4:0     STRxD/C/B/A abilitazione uscita
             0 = Px come I/O
             1 = Px come uscita PWM
```

Se i bit **STRxX** sono a **0** (default) il pin è un I/O. Portando i bit a **1** si attiva l'uscita in steering. Possono essere abilitati uno o più pin. Da notare che è possibile escludere anche **PxA**.

Il bit **STRxSYNC** consente due selezioni:

- Quando è a 0, lo steering avverrà alla fine dell'istruzione che scrive nel registro **PSTRxCON**. In questo caso, il segnale di uscita ai pin **Px<D:A>** potrà essere una forma d'onda PWM incompleta. Questa scelta è utile se c'è la necessità di rimuovere immediatamente un segnale PWM dal pin.

- Quando il bit **STRxSYNC** è 1, l'aggiornamento avverrà all'inizio del prossimo ciclo PWM. In questo caso, l'uscita PWM viene attivata o disattivata sempre con una forma d'onda PWM completa.

Il foglio dati riporta diagrammi temporali relativi alle due condizioni.

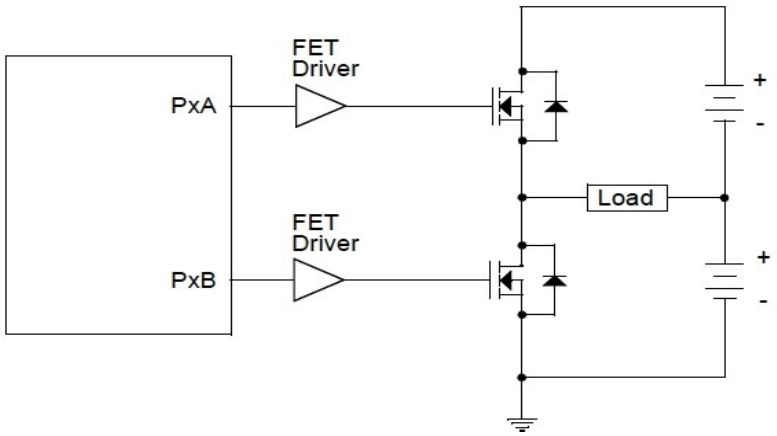
L'uso del modo singolo è analogo a tutti i tipi di moduli PWM a 8bit.

Half-bridge e Full-bridge.

Le configurazioni tipiche sono dette ponti (bridge) acausa del carico che costituisce un “ponte” tra un lato del driver e l'altro (H bridge) o tra il driver e l'alimentazione (Half-bridge).
A seconda dei componenti esterni impiegati possiamo disporre di un half-bridge (mezzo ponte) o di un full-bridge (ponte intero).
Non entriamo qui nella teoria dei driver di potenza, cosa che non fa parte di questo corso; ci limitiamo a tratteggiare degli schemi di principio, rimandando chi è interessato ad una trattazione specifica.

Half-bridge.

La modalità di controllo Half-bridge (mezzo ponte) richiede un driver esterno del genere push-pull, come rappresentato nella immagine seguente, con una alimentazione duale



Sono impegnati due segnali **PxA** e **PxB** e il PWM è applicato alle due uscite con polarità opposta. Questo è previsto per comandare una coppia complementare di driver di potenza (puh-pull).

Il nome Half-bridge deriva dal fatto che i driver di potenza sono la metà di quelli necessari ad un Full-bridge. Da notare che è richiesta una alimentazione duale per poter invertire la polarità sul carico. Questa configurazione è molto comune in SMPS e inverter e può operare anche in corrente alternata.

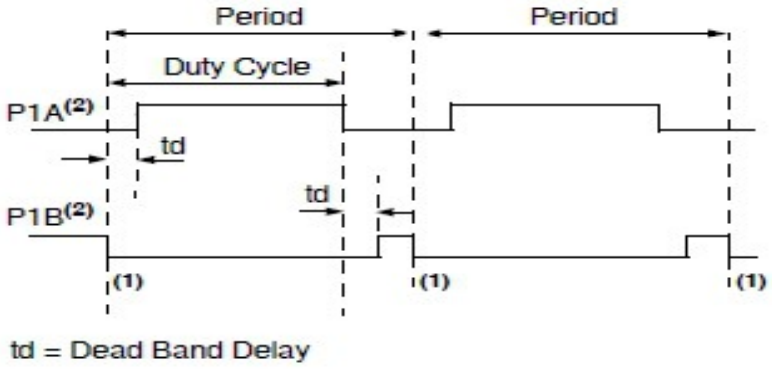
Dobbiamo agire programmando il modulo ECCP/PWM con il registro **CCPxCON**:

Modo ECCP	PxM<1:0>	PxA	PxB	PxC	PxD
Half-bridge	10	***modulated***	***modulated***	-	-

Sia **PxA** che **PxB** sono modulati dal segnale PWM (mentre **PxC** e **PxD** sono rilasciati come pin di I/O nei moduli PWM che possono servire anche Full-bridge).

In questa modalità occorre evitare che siano posti in conduzione contemporaneamente entrambi i MOSFET, cosa che genererebbe un corto circuito diretto sull'alimentazione.
Da osservare che la sovrapposizione di momenti di conduzione di entrambi i driver non dipende dal segnale PWM, ma dal tempo di spegnimento di un driver rispetto al tempo di accensione dell'altro. In genere, causa le capacità e le induttanze del circuito e dei componenti, il tempo di spegnimento risulta maggiore di quello di accensione, facendo si che per un tempo più o meno breve entrambi i transistor si trovino accesi.

Per evitare questo, è possibile programmare l'inserimento di un tempo di attesa tra le commutazioni (dead band time – tempo di banda morta), programmabile con un registro apposito;:



Questo tempo va programmato in modo da garantire lo spegnimento di un transistor prima di accendere l'altro ed è programmabile con il registro **PWMxCON**. Il ritardo avviene al passaggio del segnale dallo stato non attivo allo stato attivo. I sette bit inferiori del **PWMxCON** stabiliscono il periodo di ritardo in termini di cicli di istruzioni del microcontrollore (**TCY** o **4 TOSC**).

bit	7	6	5	4	3	2	1	0
PWMxCON	**PxRSEN**	**PxDC<6:0>**						

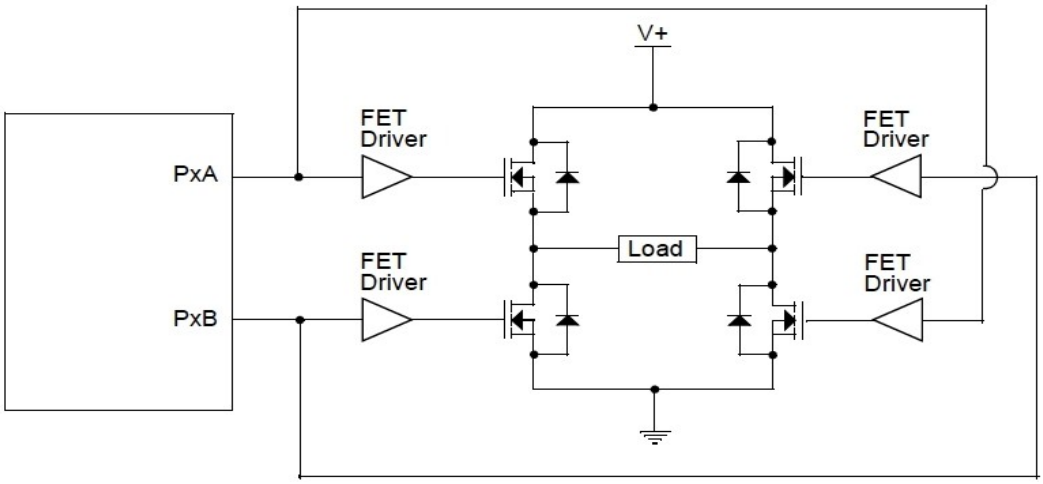
```
bit7     PxRSEN     auto restart
           1 = auto restart con cancellazione automatica del flag CCPxASE
           0 = CCPxASE deve essere cancellato da programma
bit6-0   PxDC<6:0>  Numero di cicli FOSC/4 (4 * TOSC) di ritardo nell'attivazione del
                    segnale PWM
```

Quindi:

- il ritardo dipende dal clock del sistema e varia con esso
- è possibile arrivare ad un ritardo massimo di **127 * Tcy**

La determinazione del ritardo non è quantificabile a priori, dato che sarà dipendente dalle caratteristiche hardware dei driver (è auspicabile un design in cui il tempo di spegnimento sia minimizzato, onde evitare lunghi ritardi di dead band che sono controproducenti sull'efficienza del controllo).

Dove non è possibile o necessari la doppia alimentazione, si può ricorrere ad una diversa configurazione che consente di operare in Half-bridge, ma con alimentazione singola:

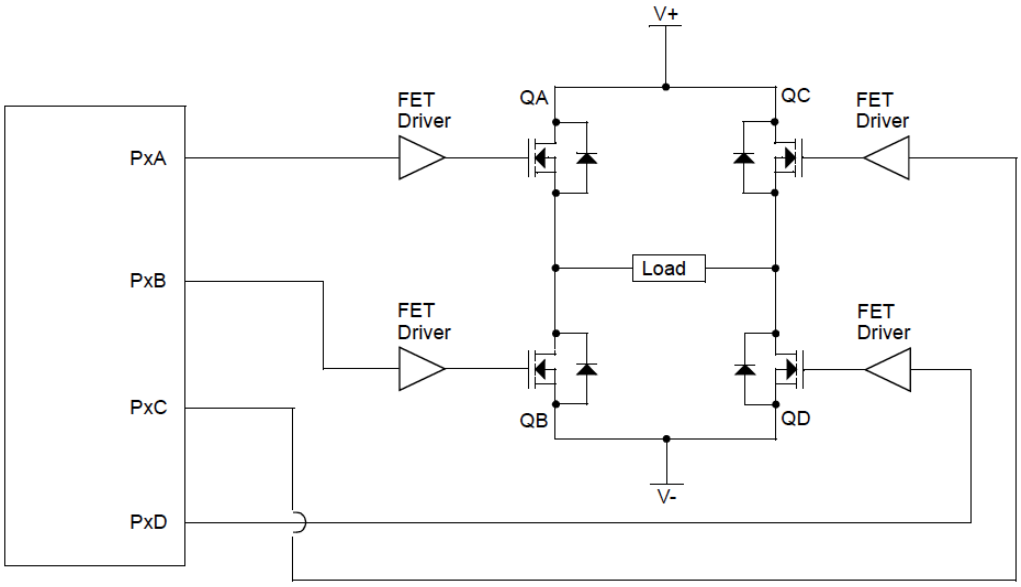


Da osservare che non si tratta di un Full-bridge, il quale richiederebbe 4 segnali di controllo: si tratta di due Half-bridge comandati dai solo due segnali (**PxA** e **PxB)** che sono incrociati sui 4 transistor del ponte. Viene detto anche **H bridge**, definizione comune al Full-bridge, con possibile confusioni.

Le due uscite PWM forniscono due segnali a polarità opposta. **PxA** accende il MOSFET superiore sinistro e l'inferiore destro, applicando una polarità al carico. Il segnale opposto accende il superiore destro e l'inferiore sinistro, applicando al carico la polarità opposta.

Full-bridge.

Utilizzando tutte e quattro le uscite, possiamo comandare in modo pieno un Full-bridge:



Osserviamo la differenza rispetto al precedente doppio Half-bridge. Qui sono in gioco 4 segnali di controllo e non solo 2.
Nel modo Forward (avanti), il pin **PxA** è pilotato al suo stato attivo e collega il carico la positivo; il pin **PxD** è modulato dal PWM e collega il carico al negativo, mentre **PxB** e **PxC** saranno allo stato inattivo, ovvero come se non ci fossero.
In modalità inversa, **PxC** è pilotato allo stato attivo, il pin **PxB** è modulato e il carico è colegato a polarità opposta alla precedente; **PxA** e **PxD** saranno nel loro stato inattivo.

Nel modo Full-Bridge, non è il duty cycle a determinare la direzione, ma il bit **PxM1** nel registro **CCPxCON,** che permette l'inversione della polarità sul carico, ad esempio avanti/indietro di un motore. Quando l'applicazione cambia questo bit di controllo, il modulo eseguirà nel prossimo ciclo PWM.

Nota:
I termini "**avanti**" e "**indietro**" sono riferiti a due situazioni opposte dei segnali di controllo del ponte, non al fatto che un motore collegato giri in senso orario piuttosto che antiorario.
La direzione di rotazione del motore comandato dal ponte rispecchia i segnali con cui il ponte è comandato.
Quindi, una corrispondenza specifica tra la direzione meccanica della rotazione e il comando elettrico del ponte dipende dai collegamenti elettrici del motore.

La sequenza di inversione avviene quattro cicli di timer prima della fine dell'attuale periodo di PWM:

- Le uscite modulate (**PxB** o **PxD**) sono posizionate nel loro stato inattivo.
- Le uscite non modulate associate (**PxA** e **PxC**) vengono commutate nell'opposta direzione.
- La modulazione PWM riprende all'inizio del prossimo periodo.

Da notare che con questo ponte è possibile l'arresto del motore senza circolazione di corrente: basta sospendere il PWM. E' compresa la possibilità di arresto con frenatura quando sono attivati entrambi i MOSFET inferiori che cortocircuitano il motore attraverso la massa.

La modalità Full-Bridge non dispone di controllo della banda morta: poiché viene modulata una uscita alla volta, un ritado alla commutazione non è generalmente richiesto. C'è una situazione dove, però, può essere necessario e che si verifica quando entrambe le seguenti condizioni sono vere:

1. La direzione dell'uscita PWM cambia quando il ciclo di funzionamento dell'uscita è pari o vicino al 100%.
2. Il tempo di spegnimento dello switch di alimentazione è maggiore del tempo di accensione dell'altro switch.

Grafici sul foglio dati rappresentano questa situazione. Se è richiesto il cambio di direzione del PWM con un elevato duty cycle, sono generalmente possibili due soluzioni per eliminare la possibilità di un corto circuito (anche se ne esistono altre): ridurre il duty cycle del PWM per un periodo prima di cambiare direzione oppure utilizzare driver degli interruttori di potenza che possano spegnersi più velocemente di quanto possano accendersi.

Il modo EPWM supporta un Auto-Shutdown che disabilita le uscite del PWM quando si verifica un evento che richiede lo spegnimento del PWM. I pin di uscita PWM vengo posti automaticamente in uno stato predeterminato dalla programmazione del registro **CCPxAS**.

bit	7	6	5	4	3	2	1	0
CCPxAS	**CCPxASE**	**CCPxAS<2:0>**			**PSSxAC<1:0>**		**PSSxBD<1:0>**	

```
bit7     CCPxASE           auto shutdown flag
          1 = si è verificato un evento di shutdown
          0 = il modulo è operativo
bit6-4   CCPxAS<2:0>  selezione sorgente shutdown
          000 = auto shutdown disabilitato
          001 = comparatore C1 livello alto
          010 = comparatore C2 livello alto
          011 =  C1 o C2 livello alto
          100 =  livello basso al pin INT
          101 =  livello basso al pin INT o C1 alto
          110 =  livello basso al pin INT o C2 alto
          111 =   livello basso al pin INT o C2 o C1
 bit3-2  PSSxAC<1:0>    stato dei pin PxA e PxC allo shutdown
          11 =  PxA e PxC in tri-state
          10 =  PxA e PxC in tri-state
          01 =  PxA e PxC=1
          00 =  PxA e PxC=0
bit3-2  PSSxBD<1:0>    stato dei pin PxB e PxD allo shutdown
          11 =  PxB e PxD in tri-state
          10 =  PxB e PxD in tri-state
          01 =  PxB e PxD=1
          00 =  PxB e PxD=0
```

Osserviamo che le fonti di shutdown, ovvero di arresto del PWM, dipendono essenzialmente dallo stato dei comparatori o dal pin di richiesta di interruzione esterna.

In effetti, i modi EPWM sono indirizzati a controllare regolazioni su motori oppure di sistemi di

riscaldamento o luci e, in generale, regolazioni di potenza, dove un errore o un guasto possono provocare danni ingenti. La funzione di shutdown è un sistema di sicurezza che blocca in modo automatico il segnale PWM quando viene rilevato un problema; ad esempio, si potrà monitorare la corrente con i comparatori, individuando una sovra corrente pericolosa, oppure rispondere ad una richiesta di arresto d'emergenza con l'ingresso INT. In entrambi i casi l'automatismo interviene istantaneamente senza richiedere istruzioni del programma.

Quando si verifica un evento di spegnimento, succedono due cose:

1. Il bit **CCPxASE** è impostato su '1' e rimane così rimanere impostato fino alla cancellazione da programma oppure al riavvio automatico. Questo consente al software di identificare l'evento.
2. I pin PWM abilitati sono posizionati in modo asincrono nei loro stati di chiusura. I pin di uscita PWM sono raggruppati in coppie [**PxA/PxC**] e [**PxB/PxD**]. Lo stato di ogni coppia di pin è determinata dai bit **PSSxAC** e **PSSxBD**. Ogni coppia di pin può essere posta in uno dei tre stati, che dipendono dall'hardware dei driver di potenza:
   - Livello 1
   - Livello 0
   - Tri-state (ad alta impedenza)

La condizione di auto spegnimento è basata sul livello del segnale, non sulla transizione: finché il livello è presente, l'auto spegnimento persisterà e il bit **CCPxASE** sarà settato. Una volta che la condizione di auto spegnimento è stata rimossa e il PWM è stato riavviato (sia attraverso il firmware o il riavvio automatico) il segnale PWM si riavvierà sempre all'inizio del prossimo periodo.
Un arresto da software può essere attivato dal programma portando a 1 il bit **CCPxASE**.

Il registro **PWMxCON** contiene sia il bit di abilitazione dell'auto restart, sia i bit di controllo del tempo della banda morta:

bit	7	6	5	4	3	2	1	0
PWMxCON	**PxRSEN**	**PxDC<6:0>**						

```
bit7      PxRSEN      auto restart
            1 = auto restart con cancellazione automatica del flag CCPxASE
            0 = CCPxASE deve essere cancellato da programma
bit6-0    PxDC<6:0>   Numero di cicli FOSC/4 (4 * TOSC) di ritardo nell'attivazione del
                      segnale PWM
```

Come per gli altri tipi di PWM, è possibile programmare diverse fonti di clock, con il registro **CCPTMRS**. Ad esempio, per 2 moduli ECCP/PWM + 2 moduli CCP/PWM integrati nel chip, si ha questo aspetto

bit	7	6	5	4	3	2	1	0
CCPTMRS	**C4TSEL<1:0>**		**C3TSEL<1:0>**		**C2TSEL<1:0>**		**C1TSEL<1:0>**	

```
bit7-6    C4TSEL<1:0>   selezione timer per CCP/PWM4       Per tutti le scelte sono:
bit5-4    C3TSEL<1:0>   selezione timer per CCP/PWM3          11 = riservato
bit3-2    C2TSEL<1:0>   selezione timer per ECCP2/PWM2        10 = Timer6
```

```
bit1-0   C1TSEL<1:0>  selezione timer per ECCP1/PWM1        01 = Timer4
                                                            00 = Timer2 (default)
```

Come per gli altri moduli PWM, si potrà usare un solo timer per più moduli oppure uno diverso per ogni modulo. Il **Timer2** è sempre il default di compatibilità con i chip meno recenti.

Per quanto riguarda le formule di calcolo, vale quanto detto per gli altri moduli PWM.

Per riassumere:

Regole generali:

- Per i chip che dispongono di più moduli PWM è necessario prestare attenzione ai nomi dei registri ed organizzare con cura questa sezione del programma in cui entrano in gioco anche i timer.
- Le uscite PWM, dove necessario, vanno assegnate ai pin con **PPS** (oppure rilocate con **APFCON** nei chip con questa opzione)
- Le uscite PWM devono avere il relativo **TRIS=0**
- Il modulo PWM deve essere abilitato (è disabilitato per default)
- Il timer che fornisce il clock deve essere attivo

Tutti gli ECCP/PWM degli Enhanced dipendono dai timer di "tipo2", ovvero Timer2, 4, 6, 8,10. Questi timer consentono di disporre di numerose opzioni e sono controllati dai registri:

- **TxCLKCON** selezione sorgente clock
- **TxHLT** selezione opzioni hardware limit
- **TxRST** selezione opzioni reset esterni
- **TxCON** selezione prescaler, postscaler, abilitazione
- **PRx** valore per la durata del periodo

Un esempio di inizializzazione per il Timer2:

```
T2CLKCON = 0;                   // Fosc/4   (default)
T2CONbits.T2CKPS = 0b00;        // prescaler 1:1   (default)
```

```
T2CONbits.T2OUTPS = 0b0000;     // postscaler 1:1 (default)
PR4 = 125-1;                    // 1ms (1KHz) @ 500kHz
TMR2ON = 1;                     // enable timer
```

Da notare che è normale che i timer associati ai moduli ECCP/PWM siano di **"tipo 2 base"**, ovvero non presentano tutte le opzioni di quelli più avanzati:

- il clock dipende solo da **Fosc/4**
- non dispongono di funzioni HLT
- hanno prescaler limitato a **1:1/1:4/1:16/1:64**

I moduli ECCP/PWM dipendono dai seguenti registri di controllo

- **CCPxCON** modo, abilitazione, allineamento bit, flag stato
- **CCPRxH/CCPxCON<5:4>** valore del duty cycle (10bit)
- **CCPTMRS** selezione del timer

Un esempio di inizializzazione per il CCP/PWM1:

```
CCPTMRSbits.CCP1TSEL = 0;       // clock da Timer2 (default)
CCP1CONbits.MODE = 0b1111;      // modo PWM
CCP1CONbits.FMT = 1;            // left aligned
CCPR1H = 0x7C;                  // preset duty 100%
CCPR1L = 0xC0;
CCP1CONbits.EN =1;              // modulo abilitato
```

L'uscita del modulo PWM va assegnata con PPS.
Ad esempio: **CCP1** out assegnato a **RA5**

```
RA5PPS = 0x0C;            // RA5->CCP1 out
```

Schema delle differenze tra i moduli PWM

Ecco un breve riassunto sintetico delle caratteristiche di controllo dei vari moduli (modo PWM)

PWM	CCP/PWM	ECCP/PWM
Per tutti, occorre definire il timer usato con il registro **CCPTMRS** (o **CCPTMRSx**). Il default abilita il **Timer2**. Il timer va programmato con i valori scelti per il periodo.		
-	Occorre scegliere il modo PWM (che non è il default) con i bit **MODE<3:0>** del registro **CCPxCON**	Occorre scegliere il modo PWM con i bit **CCPxM** (il default è disabilitato) del registro **CCPxCON**
Il valore per il duty cycle va caricato nei registri **PWMxDCH:L**, giustificati a sinistra	Il valore per il duty cycle va caricato nei registri **CCPxDCH:L** La giustificazione dei registri del duty cycle è effettuata con il bit **CCPxFMT** del registro **CCPxCON** (per default sono giustificati a destra)	Il valore per il duty cycle va caricato nei registri **CCPRxL** e nei bit **DCxB<1:0>** del registro **CCPxCON**
Occorre definire la polarità dell'uscita con il bit **PWMxPOL** del registro **PWMxCON** (il default è polarità positiva)	-	Occorre scegliere il modo enhanced (il default è Single Out) con i bit **PxM** del registro **CCPxCON**
Occorre abilitare il modulo con	Occorre abilitare il modulo con	Il modulo è abilitato con i bit

il bit **PWMxEN** del registro **PWMxCON** (il default è disabilitato)	il bit **CCPxEN** del registro **CCPxCON** (il default è disabilitato)	**CCPxM** del registro **CCPxCON** (il default è disabilitato)
-	-	I registri **CCPxAS**, **PWMxCON**, e **PSTRxCON** servono per le opzioni enhanced (disabilitati per default)