UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI TECNICA E GESTIONE DEI SISTEMI INDUSTRIALI

CORSO DI LAUREA TRIENNALE IN INGEGNERIA MECCANICA E MECCATRONICA - CURRICULUM MECCATRONICO

USO DI MICROCONTROLLORE PER IL CONTROLLO DI AZIONAMENTI DI ROBOT PER LA RIABILITAZIONE

RELATORE: **Prof. ROBERTO OBOE**

LAUREANDO: **MANUEL SPIGOLON**

ANNO ACCADEMICO: 2012/2013

INDICE

SOMMARIO

In questo elaborato si vuole sviluppare uno strumento per la consultazione per chi vuole sviluppare un sistema di controllo di un azionamento utilizzando un microcontrollore della Freescale. In particolare il microcontrollore utilizzato è stato sviluppato per le applicazione che richiedono un elevato livello di sicurezza e quindi trova una perfetta applicazione in campo medico. I robot per la riabilitazione devono essere molto stabili e sicuri perché un eventuale errore può portare a una situazione di pericolo del paziente. Verrà sviluppato come caso di studio il controllo in posizione di un motore elettrico come esempio per comprendere meglio l'utilizzo dei vari moduli interni del microcontrollore. Inoltre verranno illustrate le basi per utilizzare l'ambiente di sviluppo CodeWarrior.

CAPITOLO 1

Introduzione

1.1 Robot per la riabilitazione

La ricerca clinica ha dimostrato che i soggetti con lesione del sistema nervoso centrale (ad esempio dopo un ictus) hanno un grande potenziale di recupero se seguono una riabilitazione ripetitiva, frequente e intensa. Per questo i Robot sono molto utili per il recupero motorio, perché consentono al paziente di effettuare un allenamento ripetitivo con una intensità tarata sulle capacità residue del soggetto e di selezionare innumerevoli personalizzazioni dell'esercizio. I programmi di ricerca che hanno a che fare con lo sviluppo della robot-terapia sono aumentati nell'ultima decina di anni e continueranno a crescere perché permette anche la riduzione dei costi per la riabilitazione. Le caratteristiche essenziali dei robot impiegati per la riabilitazione sono le seguenti:

- Elevata affidabilità e sicurezza assoluta con i pazienti
- Precisione elevata
- Operatività elevate per minimizzare i costi di gestione e ammortizzare i prezzi dell'unità robotica
- Procedure automatiche di rilevazione di malfunzionamenti e autoriparazione

Questi robot hanno un interfaccia aptica per comunicare con il paziente: un dispositivo robotico studiato per interagire direttamente con l'operatore umano, che riceve in risposta sensazioni tattili, come ad esempio le forze di contatto. Semplici interfacce aptiche sono joystick con ritorno di forza. Le interfacce aptiche sono costituite da due parti principali:

- Manipolatore, che è la parte meccanica costituita da vari componenti: la base, fissata nell'ambiente di lavoro o costituita da una piattaforma mobile, una serie di link, parti rigide di collegamento, e una serie di giunti, snodi che permettono il collegamento tra i link.
- Sistema di programmazione e controllo composto da un dispositivo di calcolo al quale sono collegati sensori, attuatori, in genere elettrici o pneumatici, ed infine controllori.

Per la creazione di prototipi in laboratorio come dispositivo di calcolo si utilizza un computer che comunica con sensori e attuatori attraverso un interfaccia. Per la creazione di un sistema ad uso commerciale non si utilizza un computer ma un microcontrollore. Questa soluzione permette ugualmente la tele-riabilitazione: il dispositivo di controllo a microcontrollore collegato, per esempio tramite USB, ad un computer connesso ad internet consente ad un

fisioterapista di dialogare direttamente con l'interfaccia aptica, e quindi con il paziente, tramite il suo computer connesso ad internet, pur non trovandosi nello stesso posto.

1.2 Caso di studio

In questo elaborato si vuole sviluppare un sistema per il controllo di posizione di un motore elettrico. In particolare il motore verrà controllato con un anello PID di posizione, implementato nel microcontrollore, che genererà un riferimento di coppia per il motore. Per sapere la posizione si andrà a leggere un encoder collegato al motore. Per sviluppare il codice si è utilizzato come sistema di sviluppo CodeWarrior, sviluppato dalla Freescale per i suoi microcontrollori.

Il segnale di controllo in uscita dal microcontrollore è di tipo PWM (pulse-width modulation) e verrà adattato, con un circuito elettronico (un filtro e un circuito zero-span), per l'ingresso del convertitore. Infatti nell'ingresso del convertitore si deve avere un riferimento di tensione. Nell'uscita del convertitore si ha una corrente, proporzionale alla tensione di ingresso, che comanda il motore elettrico in corrente continua.

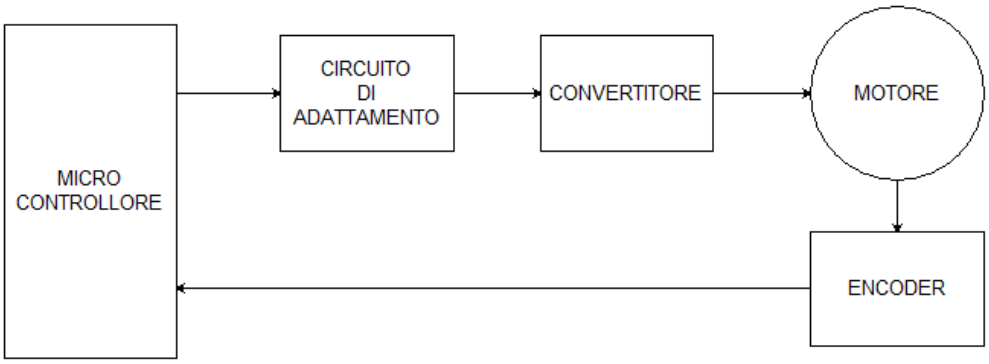


Figura 1.1: schema a blocchi caso di studio

Nel capitolo 2 vengo presentati i componenti utilizzati per il progetto con particolare attenzione il microcontrollore e i suoi moduli interni utilizzati (eTimer, flexPWM, ADC, SIUL e PIT). Nel capitolo 3 verrà fornita una descrizione dell'ambiente di sviluppo: verranno illustrate le operazioni di base per iniziare a programmare un microcontrollore utilizzando CodeWarrior. Il quarto capitolo tratterà della realizzazione del progetto: viene descritta la realizzazione del circuito di adattamento e l'implementazione software del controllo di posizione PID del motore.

CAPITOLO 2

Componenti

2.1 Tower system

La Tower System della Freescale è una piattaforma di sviluppo modulare per microcontrollori a 8, 16 o 32 bit che consente lo sviluppo avanzato attraverso una rapida realizzazione del prototipo. Sono disponibili più di cinquanta schede di sviluppo o moduli. La Tower System fornisce ai progettisti sia soluzioni per entry-level sia soluzioni per i più avanzati sviluppi del microcontrollore.

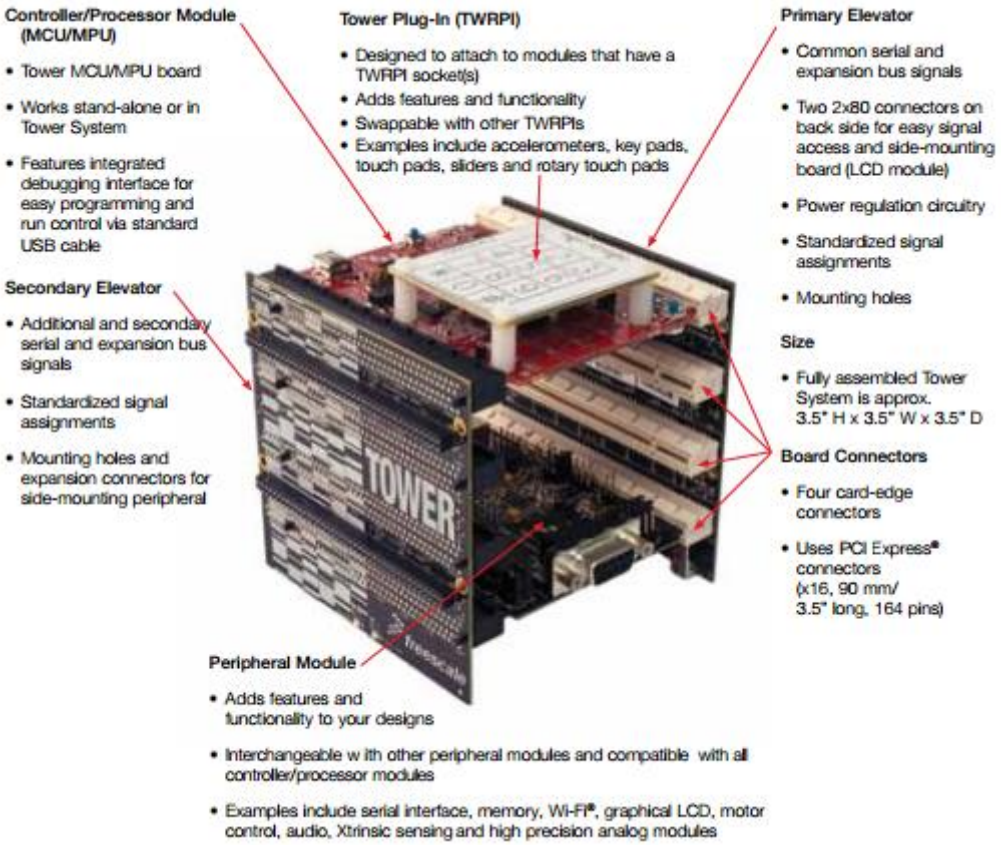


Figura 2.1: Tower System

È caratterizzata da due elevator dove, su ciascun di queste, sono presenti due connettori da 80 pin. Questi pin sono un rapido accesso ai segnali di ingresso e di uscita disponibili dal microcontrollore, per esempio i segnali PWM e gli ingressi per l'encoder. Nell'altro lato della Tower System sono presenti quattro connettori, di tipo PCI Express®, per la scheda di sviluppo e i moduli.

2.2 Microcontrollore MPC5643L

2.2.1 Caratteristiche generali

Figura 2.2: Microcontrollore MPC5643L

Il microcontrollore utilizzato è un MPC5643L della famiglia Qorivva, prodotto dalla Freescale, basato su un architettura Power Architecture® a 32 bit. Le applicazioni di questo microcontrollore sono quelle che richiedono un alto livello di sicurezza, infatti è stato progettato per affrontare in modo specifico lo standard di sicurezza ISO 26262 fino ai requisiti più severi del livello ASIL D. Per ottenere questa certificazione è stata ridotta la complessità del progetto e il numero di componenti. Le caratteristiche tecniche del dispositivo sono:

Core

- High-performance e200z4d dual core
- 32-bit Power Architecture® technology CPU
- Frequenza del core a 120 MHz
- 4 KB istruzioni cache con error detection code

Memoria disponibile

- 1 MB memoria flash con ECC
- 128 KB on-chip SRAM con ECC

Interrupts con 16 priorità

GPIO programmabili singolarmente come input, output o funzioni speciali

Tre unità di eTimer a 6 canali di uso generale

Due unità FlexPWM: 4 canali a 16 bit per modulo

Comunicazioni

- 2 canali LINFlexD

- 3 canali DSPI
- 2 interfacce FlexCAN (2.0B Active) con 32 message objects
- Modulo FlexRay (V2.1 Rev. A) con 2 canali, 64 message buffers e velocità di trasferimento superiore a 10 Mbit/s

Due convertitori analogici digitali (ADC) a 12-bit

- 16 canali di input
- Programmabile cross triggering unit (CTU) per sincronizzare la conversione degli ADC con timer e PWM

2.2.2 Modalità di funzionamento

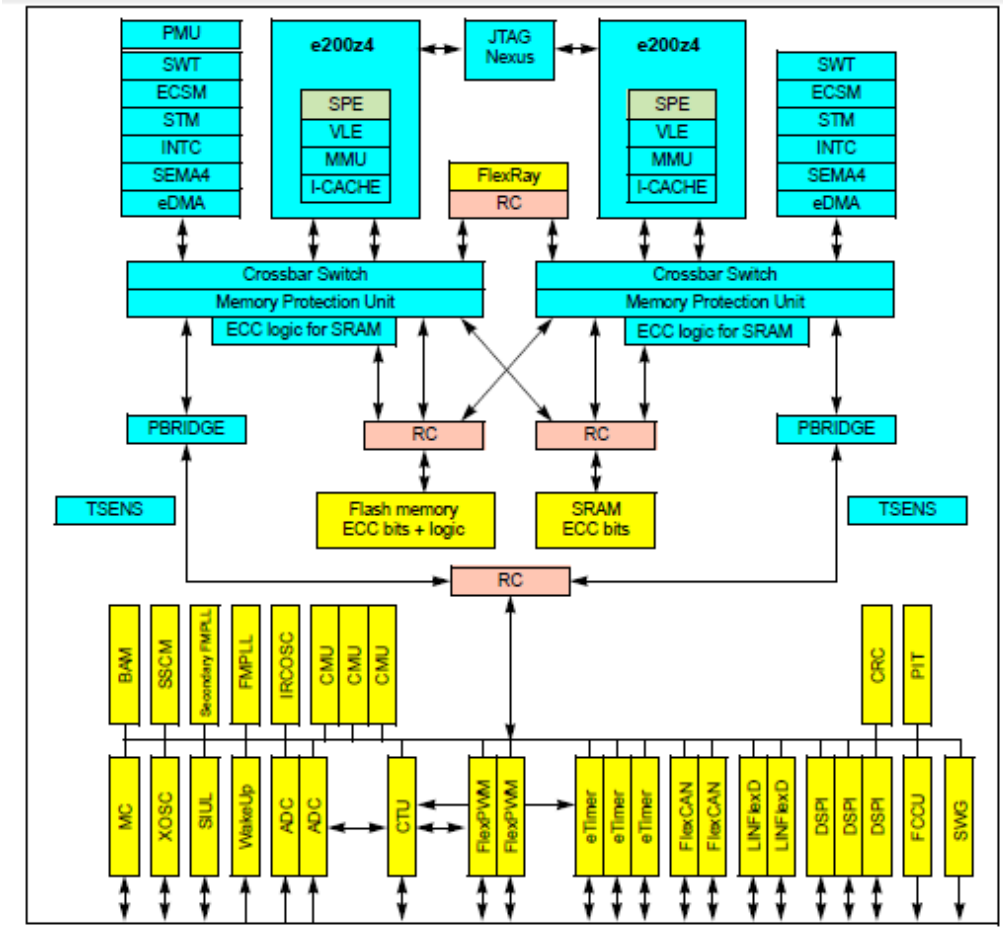
Il microcontrollore ha due modalità di funzionamento: Lock step mode (LSM) e Decoupled parallel mode (DPM).

La modalità di funzionamento Lock step mode utilizza i due core del microcontrollore eseguendo le istruzioni in modo ridondante e sincrono. Per questo motivo essa viene preferita nelle applicazioni che richiedono un maggiore grado di sicurezza. In questa modalità è importante la Sphere of Replication (Sor), in cui tutti gli elementi hardware al suo interno sono stati replicati per motivi di sicurezza. Il Sor contiene i seguenti moduli replicati:

- e200z4 core (incluso Memory Management Unit)
- Enhanced Direct Memory Access (eDMA)
- Interrupt Controller (INTC)
- Crossbar Switch (XBAR)
- Memory Protection Unit (MPU)
- Flash Memory Controller (PFlashC)
- Static RAM Controller (SRAMC)
- System Timer Module (STM)
- Software Watchdog Timer (SWT)
- Peripheral Bridge (PBRIDGE)

Ciascun elemento esegue le stesse operazioni che poi verranno controllate alla fine dal Sor. Se c'è un errore questo non propaga al di fuori del Sor e quindi non influenza le periferiche, non creando situazioni di pericolo. In Decoupled parallel mode ogni core funziona in modo indipendente dall'altro per incrementare le prestazioni. Infatti, alla stessa frequenza, le performance in questa modalità sono 1,6 volte superiori della modalità Lock step mode.

La struttura a blocchi del microcontrollore è visibile in figura 2. In particolare si nota la struttura simmetrica del dual-core e i blocchi in azzurro rappresentano il Sor. Il Sor è collegato alla memoria RAM, alla memoria Flash e al bus delle periferiche attraverso dei blocchi RC (Redundancy Checker): questi fanno in modo che gli errori non si propaghino all'esterno nella modalità LSM, invece nella modalità DPM questi blocchi sono disabilitati. In basso sono rappresentati i blocchi delle periferiche come l'eTimer, FlexPWM, l'ADC e il LineFlex.



ADC – Analog-to-Digital Converter
BAM – Boot Assist Module
CMU – Clock Monitoring Unit
CRC – Cyclic Redundancy Check unit
CTU – Cross Triggering Unit
DSPI – Serial Peripherals Interface
ECC – Error Correction Code
ECSM – Error Correction Status Module
eDMA – Enhanced Direct Memory Access controller
FCCU – Fault Collection and Control Unit
FlexCAN – Controller Area Network controller
FMPLL – Frequency Modulated Phase Locked Loop
INTC – Interrupt Controller
IRCOSC – Internal RC Oscillator
JTAG – Joint Test Action Group interface
LINFlexD – LIN controller with DMA support
MC – Mode Entry, Clock, Reset, & Power
PBRIDGE – Peripheral bridge
PIT – Periodic Interrupt Timer
PMU – Power Management Unit
RC – Redundancy Checker
RTC – Real Time Clock
SEMA4 – Semaphore Unit
SIUL – System Integration Unit Lite
SSCM – System Status and Configuration Module
STM – System Timer Module
SWG – Sine Wave Generator
SWT – Software Watchdog Timer
TSENS – Temperature Sensor
XOSC – Crystal Oscillator

Figura 2.3: Schema a blocchi microcontrollore MPC5643L

2.2.2 Architettura Clock

Nel microcontrollore sono presenti diversi clock selezionabili attraverso il Clock Generation Module (MC_CGM): un oscillatore (XOSC) e un oscillatore RC (IRCOSC). Sono presenti, inoltre, due FMPLL (Frequency-Modulated Phase-Locked Loop) che sono in grado di generare un clock ad alta frequenza con in ingresso un clock tra 4 e 40MHz.

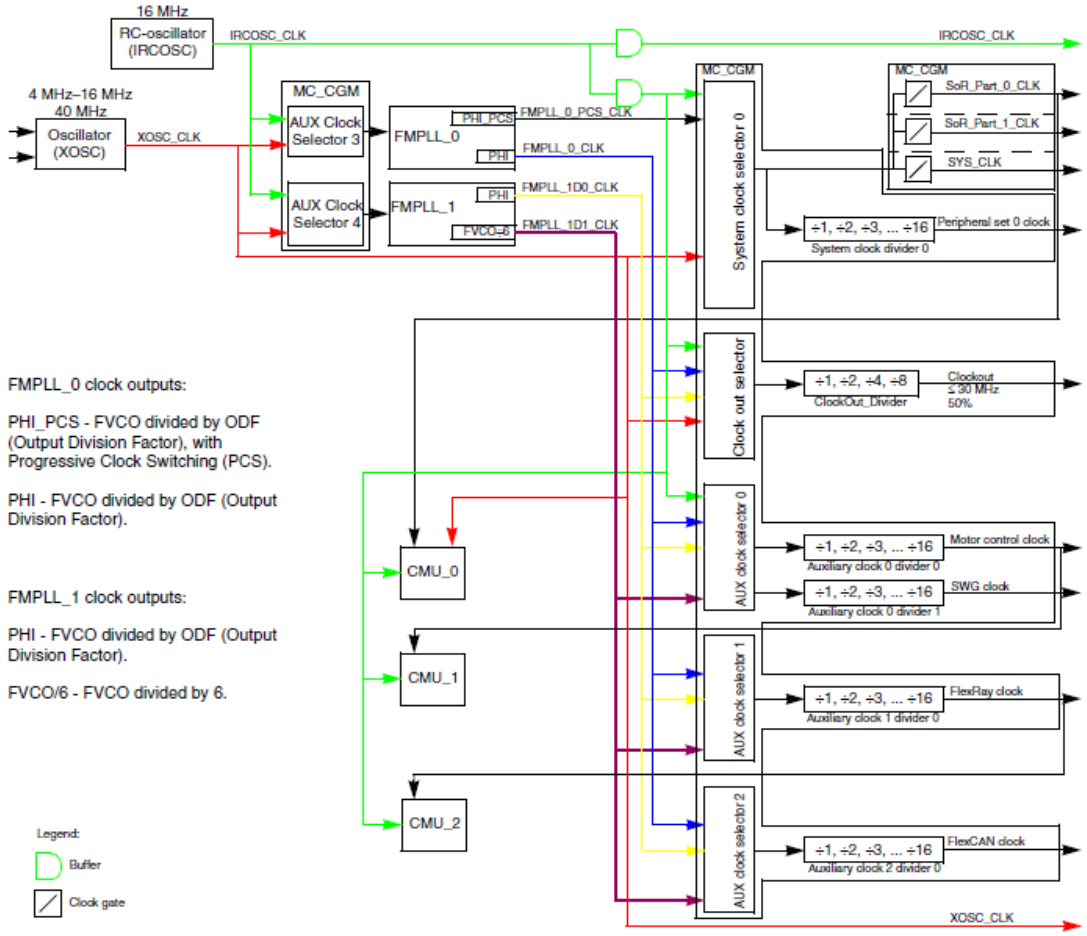


Figura 2.4: schema a blocchi clock del microcontrollore

Un FMPLL viene utilizzato dal system clock per la modulazione in frequenza mentre l'altro viene utilizzato per le periferiche motor control.

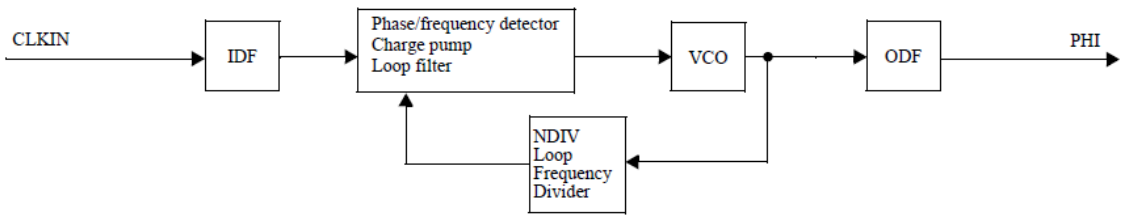


Figura 2.5: schema a blocchi FMPLL

Come si vede dalla figura 2.5 il clock in ingresso viene modificato tramite i parametri IDF, ODF e LDF che vengono impostati nei registri Control Register (CR). La frequenza del clock FMPLL in uscita (PHI) si modifica secondo la formula:

$$PHI = \frac{XOSC \times LDF}{IDF \times ODF}$$

Bisogna rispettare anche una condizione che riguarda la frequenza Fvco che deve essere compresa tra 256 e 512 MHz. Questa frequenza si ricava con la seguente formula:

$$f_{vco} = \frac{CLKIN}{IDF} \times NDIV$$

Nel progetto il clock FMPLL si è configurato per andare alla massima frequenza possibile cioè 120MHz. Per questa configurazione sono stati impostati i seguenti parametri:

XOSC = 40 MHz

NDIV = 96

IDF = 8

ODF = 4

In questo modo è stato rispettato Fvco che risulta 480MHz.

Per cambiare il clock FMPLL a 120MHz si è modificato la funzione InitModesAndClks() dove si inizializzano le configurazioni e i clock di base. In particolare si sono modificati i seguenti registri:

```
/* 120 MHz */
CGM.FMPLL[0].CR.B.IDF = 0x7; /* FMPLL0 IDF=7 --> divide by 8 */
CGM.FMPLL[0].CR.B.ODF = 0x1; /* FMPLL0 ODF=1 --> divide by 4*/
CGM.FMPLL[0].CR.B.NDIV = 96; /* FMPLL0 NDIV=96 --> divide by 96 */
```

2.2.3 System Integration Unit Lite (SIUL)

Il modulo SIUL viene utilizzato per la gestione e la configurazione dei pad, delle porte, degli ingressi/uscite GPIO (general-purpose input and output) e degli interrupt esterni configurabili come eventi di trigger. In particolare nel modulo si configurano i pin del microcontrollore: infatti dispone di 121 pin che possono essere configurati come input/output. Di questi 121 pin 22 possono essere configurati con funzionalità General Purpose Input (GPI) e 99 con funzionalità General Purpose Input/Ouput (GPIO).

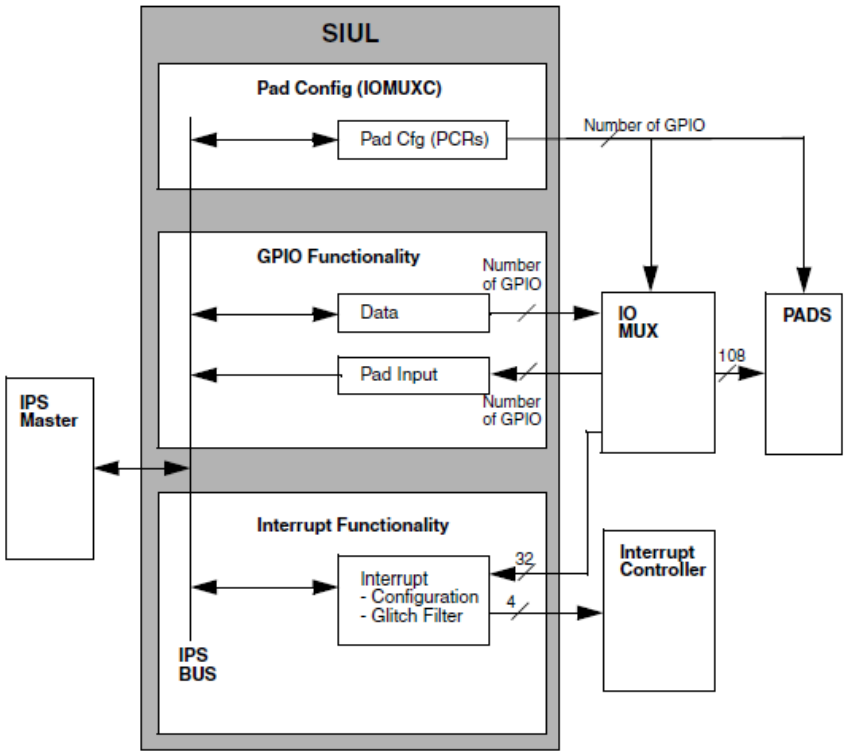


Figura 2.6: schema a blocchi modulo SIUL

In questo modulo si utilizzano principalmente due registri: il PCR (Pad Configuration Registers) dove si configura il pad come ingresso/uscita e le caratteristiche elettriche, e il PSMI (Pad Selection for Multiplexed Inputs) dove si seleziona quale funzione di input ha preso in considerazione.

2.2.4 Enhanced Motor Control Timer (eTimer)

Il microcontrollore è provvisto di 3 moduli eTimer ciascuno con 6 canali. I moduli eTimer sono contatori a 16 bit che permettono di contare a una frequenza massima di 120MHz. Se si utilizza il clock interno la velocità massima di conteggio è uguale a quella del clock, mentre è di metà clock se questo è esterno.

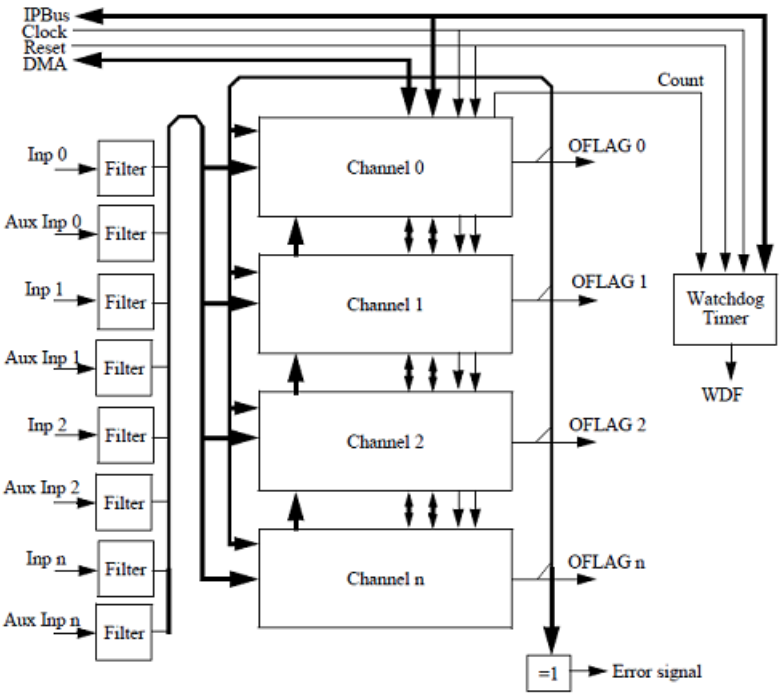


Figura 2.7: schema a blocchi eTimer

Ciascun canale ha un contatore, un prescaler, un load register, un hold register, due capture registers, due compare registers, due compare load registers e quattro registri di controllo. Il load register inizializza il valore del contatore. I compare load register servono a caricare il contatore con i valori impostati quando questo ha raggiunto i valori caricati sui compare register. L'hold register cattura il corrente valore del contatore quando gli altri contatori vengono letti. I capture register abilitano che un segnale esterno prenda il valore corrente del contatore. Il clock del contatore è preso dal peripheral clock e può essere modificato utilizzando il prescaler.

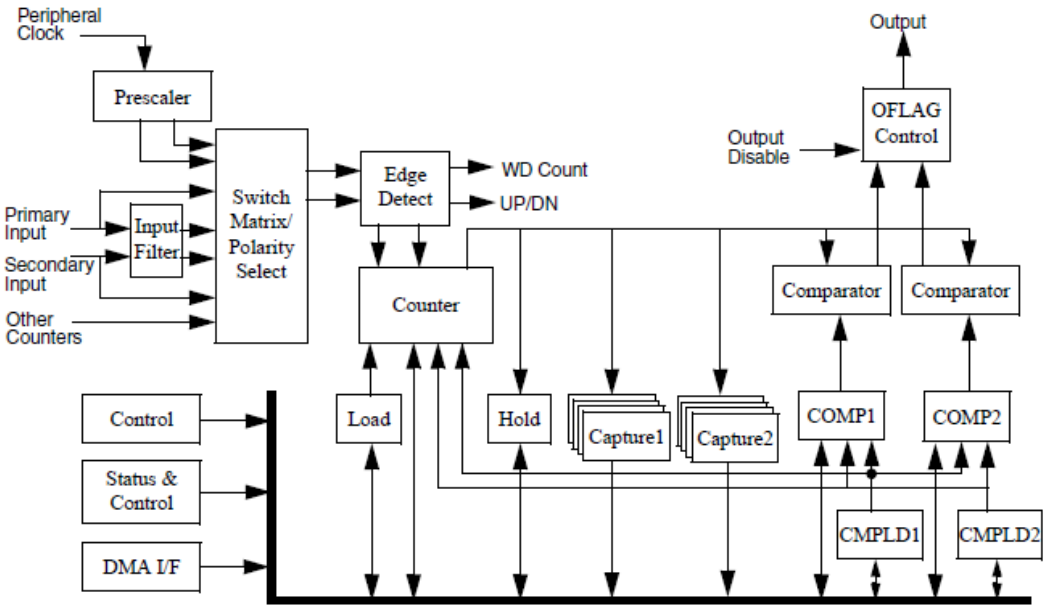


Figura 2.8: schema a blocchi di un canale di un eTimer

Le modalità di conteggio sono molteplici e vengono impostate in CNTMODE nel control register 1 (Tabella 2.1).

Value	Meaning
000	No Operation
001	Count rising edges of primary source[1]
010	Count rising and falling edges of primary source[2]
011	Count rising edges of primary source while secondary input high active
100	Quadrature count mode, uses primary and secondary sources
101	Count primary source rising edges, secondary source specifies direction (1 = minus)[3]
110	Edge of secondary source triggers primary count till compare
111	Cascaded counter mode, up/down[4]

Tabella 2.1: CNTMODE

In particolare per l'acquisizione del segnale dell'encoder si utilizza la quadrature count mode. In questa modalità si riesce anche a conoscere l'informazione sulla direzione dell'encoder. Infatti mettendo come primary source il canale A dell'encoder e come secondary source il canale B, il contatore si incrementa solo se i fronti sono alternati. Se trova due fronti consecutivi dello stesso canale inizia a decrementare il contatore perché significa che l'encoder ha invertito il moto.

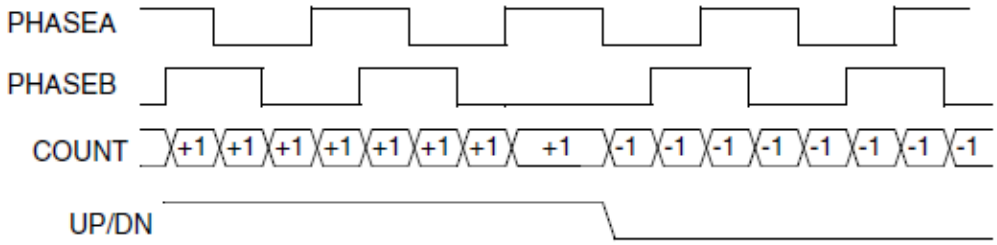


Figura 2.10: quadrature count mode

Un'altra modalità che è stata utilizzata è stata il master/slave. Ogni canale del timer può essere assegnato come Master (MSTR = 1 del registro CTRL2), e fa in modo che gli altri canali del modulo (che sono stati settati con COINIT = 1 nel registro CTRL2) possano venir re-inizializzati oppure forzare il segnale d'uscita OFLAG (COFRC = 1 nel registro CTRL2). Le modalità operative della reinizializzazione e del segnale OFLAG, impostabili dal campo OUTMODE del registro CTRL2, sono le seguenti:

Value	Meaning
0000	Software controlled
0001	Clear OFLAG output on successful compare (COMP1 or COMP2)
0010	Set OFLAG output on successful compare (COMP1 or COMP2)
0011	Toggle OFLAG output on successful compare (COMP1 or COMP2)
0100	Toggle OFLAG output using alternating compare registers
0101	Set on compare with COMP1, cleared on secondary source input edge
0110	Set on compare with COMP2, cleared on secondary source input edge
0111	Set on compare, cleared on counter roll-over
1000	Set on successful compare on COMP1, clear on successful compare on COMP2
1001	Asserted while counter is active, cleared when counter is stopped.
1010	Asserted when counting up, cleared when counting down.
1011	Reserved
1100	Reserved
1101	Reserved

Tabella 2.2: OUTMODE

Il modulo eTimer può generare un interrupt in diversi modi (Tabella 2.3). Per abilitare gli interrupt si utilizza il registro INTDMA (Interrupt and DMA Enable), mentre le relative flag si trovano nel registro STS (Status Register).

Core interrupt	Interrupt flag	Interrupt enable	Name	Description
TC0IR–TC5IR[1]	TCF	TCFIE	Compare interrupt	Compare of counter and related compare register
	TCF1	TCF1IE	Compare 1 interrupt	Compare of the counter and COMP1 register
	TCF2	TCF2IE	Compare 2 interrupt	Compare of the counter and COMP2 register
	TOF	TOFIE	Overflow interrupt	Generated on counter roll-over or roll-under
	IELF	IELFIE	Input Low Edge interrupt	Falling edge of the secondary input signal
	IEHF	IEHFIE	Input High Edge interrupt	Rising edge of the secondary input signal
	ICF1	ICF1IE	Input Capture 1 interrupt	Input capture event for CAPT1
	ICF2	ICF2IE	Input Capture 2 interrupt	Input capture event for CAPT2

Tabella 2.3: interrupt

2.2.5 Flexible Motor Control Pulse Width Modulator Module (FlexPWM)

Il microcontrollore contiene 2 moduli FlexPWM ciascuno con 4 submoduli. Il segnale PWM ottenuto è in grado di controllare la maggior parte dei motori come il motore a induzione AC (ACIM), motore a magneti permanenti AC (PMAC), il motore DC (BDC) e brushless DC (BLDC), motori a riluttanza variabile (VRM) e i motori a passo. Le caratteristiche del modulo sono:

- Risoluzione a 16 bit per PWM centrata, allineata al fronte o asimmetrica
- Frequenza massima di funzionamento di 120MHz
- L'uscita PWM può essere a canali indipendenti o a coppie complementari
- Per la generazione del segnale può accettare numeri con segno
- Controllo indipendente di entrambi i fronti di ciascun segnale PWM
- Sincronizzazione con un hardware esterno o con un altro segnale PWM
- Programmazione indipendente della polarità del segnale PWM
- Inserimento indipendente del deadtime
- Le uscite possono essere programmate per cambiare su un evento "Force out"

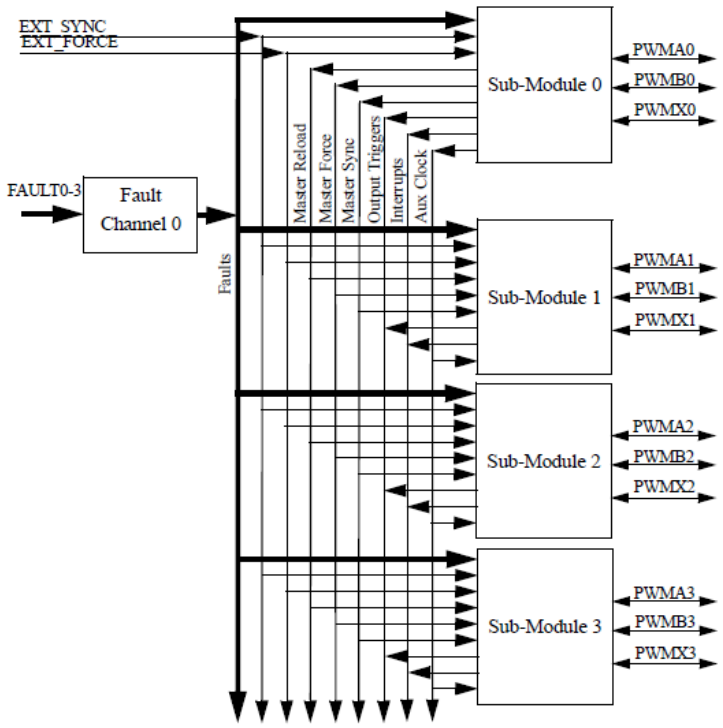


Figura 2.11: schema a blocchi modulo FlexPWM

Per la generazione dei segnali PWM il microcontrollore utilizza dei timer che sono interni ai submoduli. Questi contatori possono contare solamente in avanti e iniziano dal valore che si carica sul registro INIT. Il valore massimo si carica sul registro VAL1 e, se il valore di INIT è il complemento a 2 del valore di VAL1, il generatore di PWM funziona in modalità con segno.

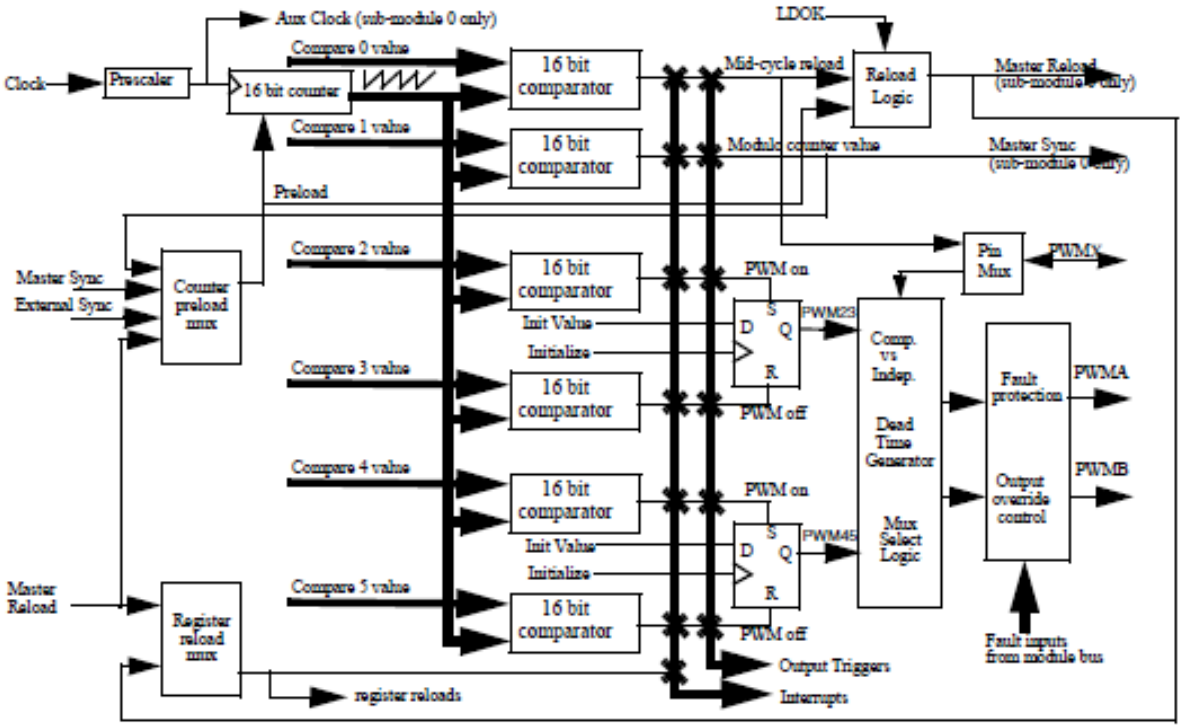


Figura 2.12: schema a blocchi submodulo FlexPWM

I valori caricati sui comparatori del submodulo sono quelli che fanno in modo di cambiare il fronte del segnale PWM. I submoduli generano due segnali PWM: i valori dei comparatori 2 e 3 (VAL2 e VAL3) comandano il primo segnale (PWMA), mentre i comparatori 4 e 5 (VAL4 e VAL5) comandano il secondo (PWMB).

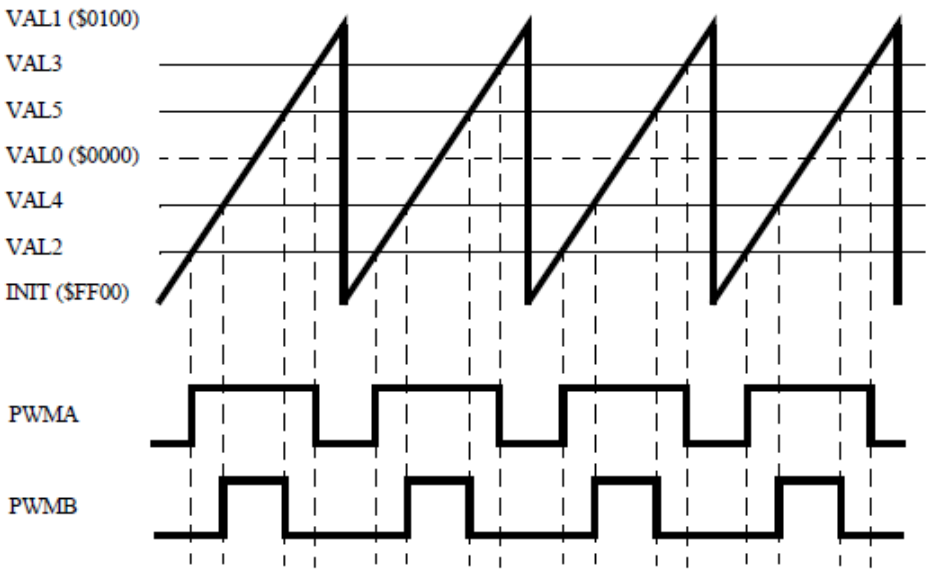


Figura 2.13: generazione segnale PWM centrato

È possibile utilizzare l'allineamento al fronte: se si imposta l'inizio del fronte del segnale PWM uguale al valore iniziale del conteggio (INIT) per cambiare il duty cycle basta cambiare solamente il valore del fronte di discesa. In questo modo, se si utilizza la modalità con segno (INIT e VAL1 dello stesso valore ma con segno opposto), a duty cycle inferiori al 50% corrispondono tensioni negative mentre ai duty cycle maggiori del 50% corrispondono tensioni positivi. Inoltre c'è una diretta proporzionalità tra il valore di discesa del fronte e la tensione al motore, incluso il segno.

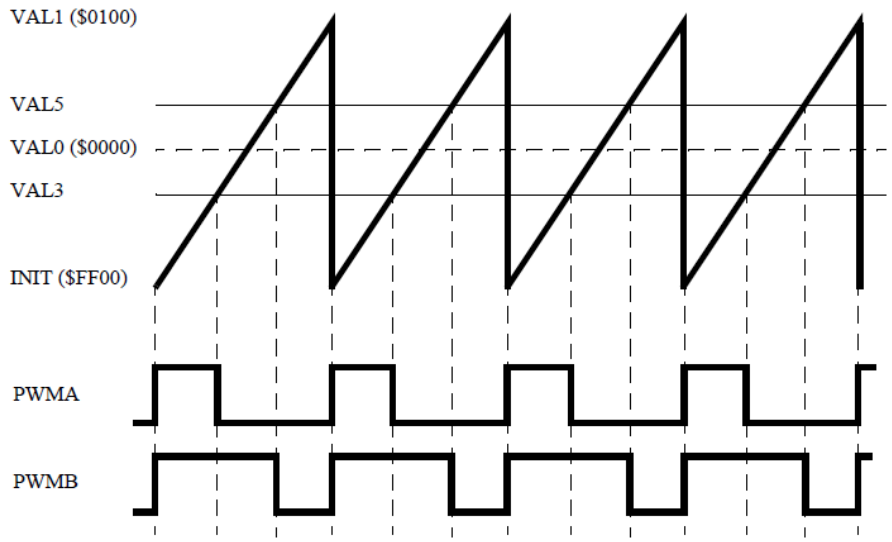


Figura 2.14: segnale PWM allineato al fronte

I valori di INIT e di VAL1 determinano anche la frequenza del segnale PWM in uscita. Per esempio se si fa funzionare il contatore a 12 bit la frequenza del segnale d'uscita risulta 29,296KHz. La frequenza della PWM si calcola come:

$$F_{PWM} = \frac{F_{CLOCK}}{n}$$

Dove n rappresenta la dimensione del contatore.

2.2.6 Analog-to-Digital Converter (ADC)

Il convertitore analogico digitale presente nel microcontrollore è di tipo SAR (Successive Approximation register). Nel dispositivo sono presenti 2 ADC con 9 canali esterni, 3 interni e 4 condivisi tra i 2 ADC, con un totale di 16 canali. Le caratteristiche dell'ADC sono:

- Risoluzione 12 bit
- Frequenza massima di clock: da 20MHz a 60MHz in base alle configurazioni
- Frequenza minima di clock: da 3MHz a 12MHz in base alle configurazioni
- Tempo di campionamento a 60MHz: programmabile da 383ns a 4.25us
- Tempo di conversione a 60MHz: programmabile da 1us a 4.87us (tempo di campionamento incluso)
- Tensione di riferimento da 3V a 5.5V

Ci sono due modalità di conversione disponibili: normal conversion e injected conversion. Nella prima modalità la conversione viene effettuata per i canali che sono stati abilitati nel registro NCMR, e quando viene fatta partire la conversione settando a 1 il campo NSTART del registro

MCR, vengono convertiti in sequenza i canali abilitati. Nella modalità injected conversion, durante una normal conversion, viene interrotta la sequenza dei canali che stanno per essere campionati facendo campionare i canali che sono stati abilitati nella injected conversion. Finito di campionare i canali abilitati nella injected conversion il convertitore ADC continua a campionare i canali abilitati nella normal mode.

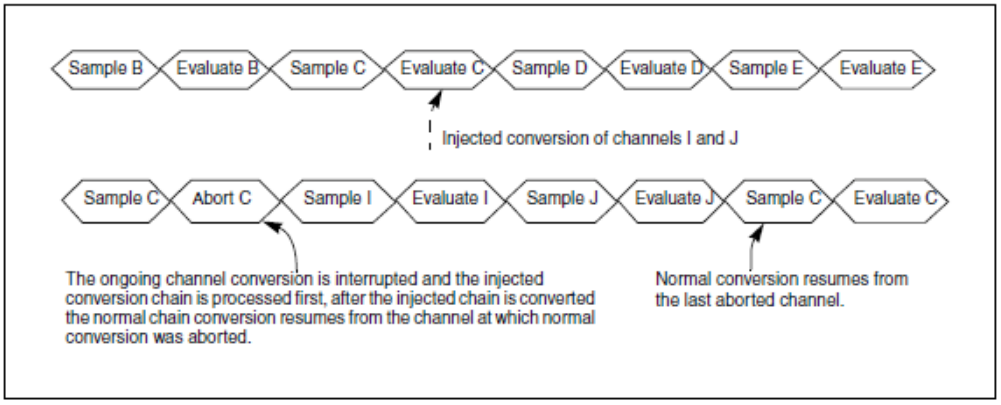


Figura 2.15: injected conversion

2.2.7 Periodic Interrupt Timer (PIT)

Il modulo PIT genera degli interrupt in modo periodico. Questi interrupt vengono generati quando il timer, che conta decrementando il valore caricato dall'utente, arriva a 0. I timer presente in questo modulo sono 4.

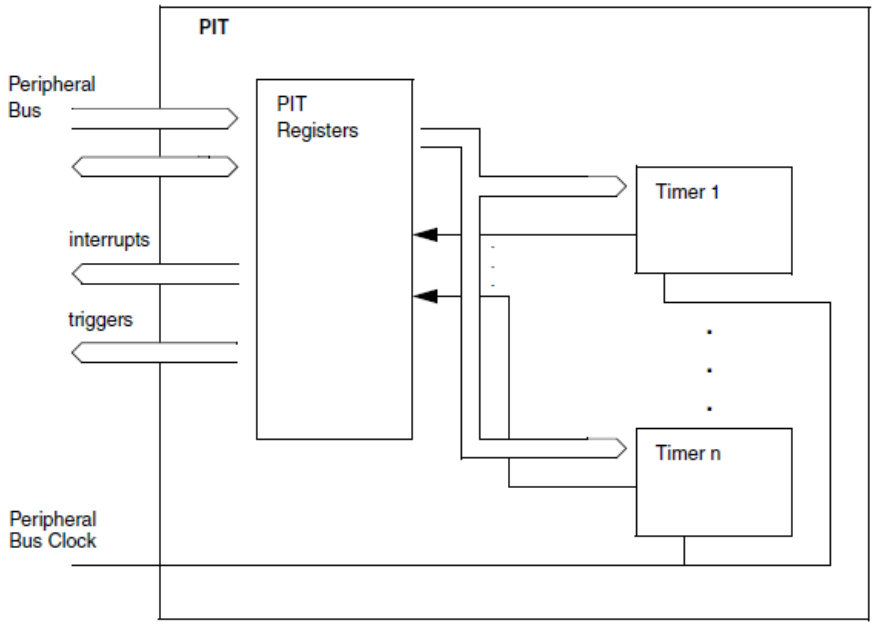


Figura 2.16: schema a blocchi modulo PIT

Per abilitare il modulo si mette a zero il campo MDIS del registro PITMCR mentre per abilitare il timer si mette a 1 il campo TIE del registro TCTRL. Per far partire il timer si setta a 1 il TEN del registro TCTRL. Il valore del timer iniziale si carica sul registro LDVAL, e si calcola come (periodo/periodo del clock) -1. Quando l'interrupt è avvenuto, e deve ripartire il timer, bisogna cancellare la flag TIF presente nel registro TFLG ponendola a 1.

Di seguito viene riportato un esempio per configurare il modulo per generare un interrupt ogni 1ms. Il peripheral bus clock è a 120MHz quindi il valore da caricare nel contatore è di 119999.

```
//Enable PIT
PIT.PITMCR.R=0x00;

//Enable PIT0 every 1 ms
PIT.LDVAL0.R=0x0001D4BF; //Load value 119999
PIT.TCTRL0.B.TIE=0x1;
PIT.TCTRL0.B.TEN=0x1;
```

2.3 Tower Serial Module

Nella Tower System in dotazione si è utilizzato il modulo seriale. Questo modulo permette ai progettisti di connettersi con la Tower System attraverso USB, Ethernet, CAN o RS232/485.

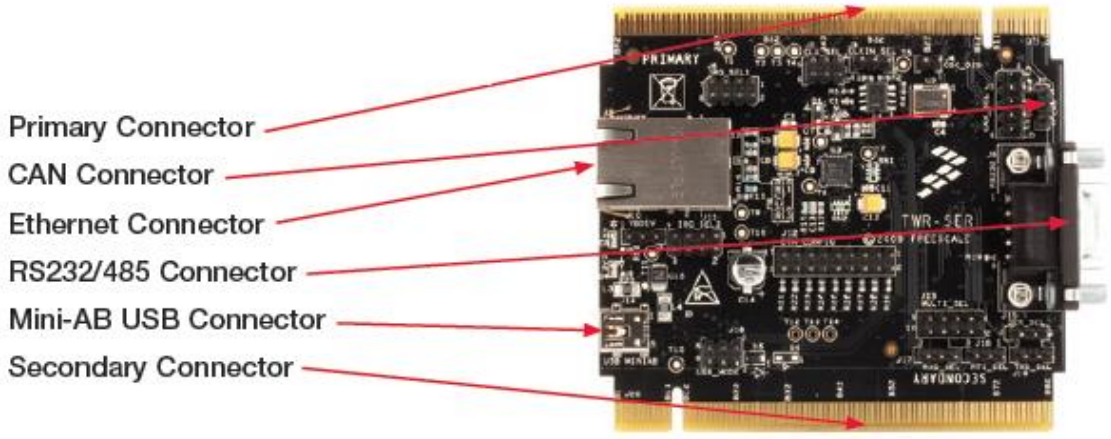


Figura 2.17: serial module

2.4 Encoder

L’encoder è un trasduttore di posizione angolare: un dispositivo elettromeccanico che converte la posizione angolare del suo asse rotante in segnali elettrici digitali.

Gli encoder possono essere di due tipi:

- Incrementali quando i segnali d'uscita sono proporzionali in modo incrementale allo spostamento effettuato.
- Assoluti quando ad ogni posizione dell'albero corrisponde un valore ben definito.

L’encoder di tipo incrementale segnalano unicamente gli incrementi (variazioni) rilevabili rispetto a un'altra posizione assunta come riferimento. Esso è costituito da un disco trasparente sul cui bordo sono stati ricavati dei settori opachi ugualmente distanziati. Ai lati del disco sono presenti, da una parte un fotoemettitore e dall’altra un fotorivelatore.

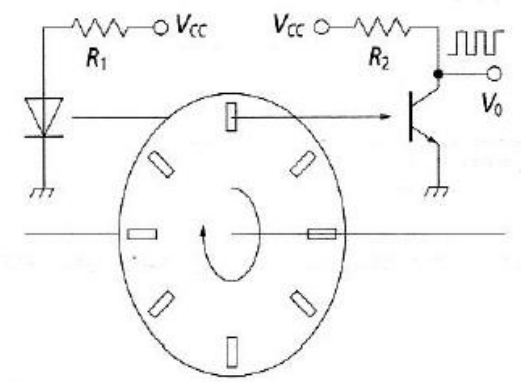


Figura 2.18: schema encoder incrementale

Viene generarano un treno di impulsi il cui numero è pari al numero delle zone trasparenti, alternate alle scure, intercettate dal blocco emettitore-ricevitore. Due corone concentriche poste sul disco una sopra l’altra, ma sfasate di metà passo, permettono inoltre all'encoder di capire quando il disco gira in senso orario e quando antiorario.

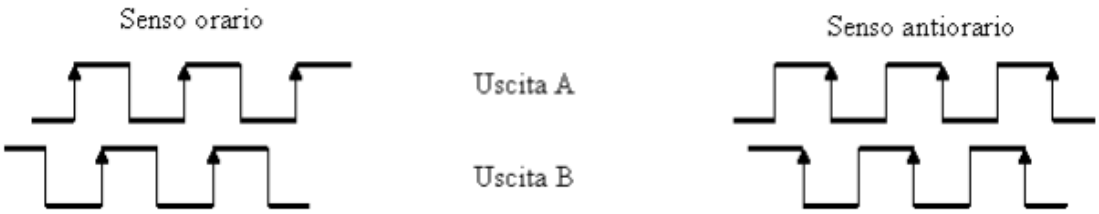


Figura 2.19: uscite encoder con due dischi

Il disco viene calettato sull'albero dell'apparecchiatura di cui si vuole rilevare lo spostamento angolare. Di conseguenza ad ogni spostamento dell'albero si ha uno spostamento uguale dell'encoder.

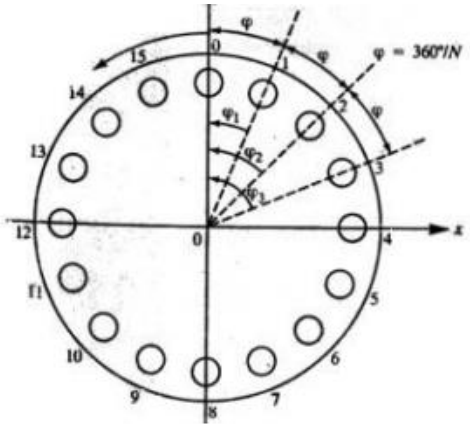


Figura 2.20: disco encoder

Il conteggio di questi impulsi consente di individuare la rotazione compiuta dal disco ed il corrispondente spostamento, infatti ad ogni impulso generato significa che il disco si è girato di $\phi=2\pi/N$, dove N è il numero di fori.

L'encoder utilizzato è un Eltra modello EH30M500Z5P6X3PR2. Le caratteristiche tecniche di questo encoder sono:

- Risoluzione di 500 impulsi/giro
- Dotato di impulso di zero (impulso che rivela quando l'encoder ha compiuto un giro)
- Alimentazione con 5V DC
- Velocità massima di 3000 rpm

Foto 2.21: encoder Eltra EH30

2.5 Convertitore

Il convertitore utilizzato è prodotto dalla speeder motion e il modello è MicroStar B. Questo convertitore è adatto al pilotaggio sia di motori BRUSHLESS che di motori in corrente continua DC. Ha in dotazione diverse modalità operative e, in questo progetto, verrà utilizzato come convertitore in controllo di coppia: questo significa che il motore verrà controllato fornendo un riferimento analogico di coppia in ingresso. La tensione di alimentazione è tra 20 e 84 Vdc, nominale 60Vdc, e le correnti disponibili sono nominale di 10A e di picco 20A. La tensione massima degli ingressi analogici è di ±10Vdc.

Comando in corrente del convertitore

Applicando una tensione proveniente, nel nostro caso dal circuito zero-span, come da figura 2.22, si può comandare il convertitore in coppia.

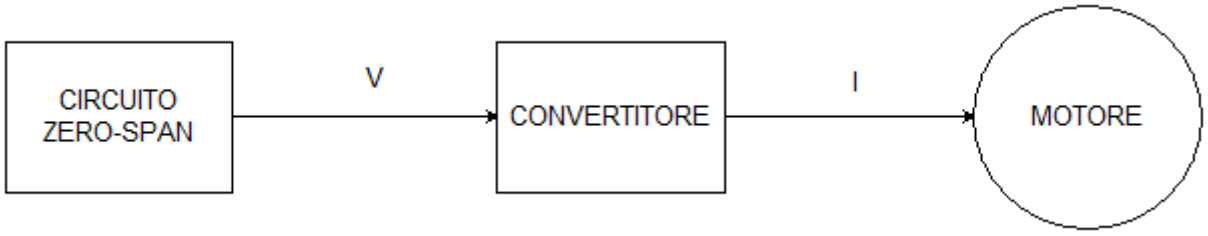


Figura 2.22: schema a blocchi conversione

Applicando un segnale di ±10V massimi il convertitore fornirà corrispondentemente la corrente di picco positiva o negativa. La formula per determinare il valore di Ving da applicare in TPRC per ottenere la corrente richiesta è la seguente:

$$V(TPRC) = \frac{10}{20} \times corrente\ richiesta$$

2.6 Motore elettrico

Per questo progetto è stato utilizzato un motore elettrico in corrente continua, prodotto dalla speeder motion, modello MB057DG218+. I dati di targa sono i seguenti:

MOTOR PARAMETERS @25°C		
CONTINUOUS STALL TORQUE	Nm	0.2
PEAK STALL TORQUE	Nm	1.05
CONTINUOUS STALL CURRENT	A	3.0
MAXIMUM PULSE CURRENT	A	14.7
MAXIMUM TERMINAL VOLTAGE	V	60
MAXIMUM SPEED	RPM	6000
MECHANICAL DATA		
ROTOR MOMENT OF INERTIA	Kg m^2	2.7*10^-5
MECHANICAL TIME CONSTANT	ms	8.4
MOTOR MASS	Kg	1.0
THERMAL DATA		
THERMAL RESISTANCE (ARMATURE TO AMBIENT)	°C/W	5
MAXIMUM ARMATURE TEM.	°C	155
WINDING SPECIFICATIONS		
TORQUE CONSTANT KT	Nm/A	0.071
VOLTAGE CONSTANT(BACK EMF)	V/KRPM	7.41
ARMATURE RESISTANCE	OHMS	1.24
TERMINAL RESISTANCE	OHMS	1.55
ARMATURE INDUCTANCE	mH	3.39
ELECTRICAL TIME CONSTANT	ms	2.1

Figura 2.23: motore MB057DG218+ con encoder

CAPITOLO 3

Ambiente di sviluppo: CodeWarrior

CodeWarrior Development Studio for Microcontrollers V10.x è un ambiente di sviluppo per i microcontrollori prodotti dalla Freescale delle famiglie ColdFire®, ColdFire+, DSC, Kinetis, Qorivva, RS08 and S08. CodeWarrior è basato sulla ben più nota piattaforma di sviluppo Eclipse, e offre strumenti ottimizzati per sfruttare appieno il microcontrollore Freescale selezionato per lo sviluppo del progetto.

Una volta aperto CodeWarrior si presenterà la schermata di figura 3.1.

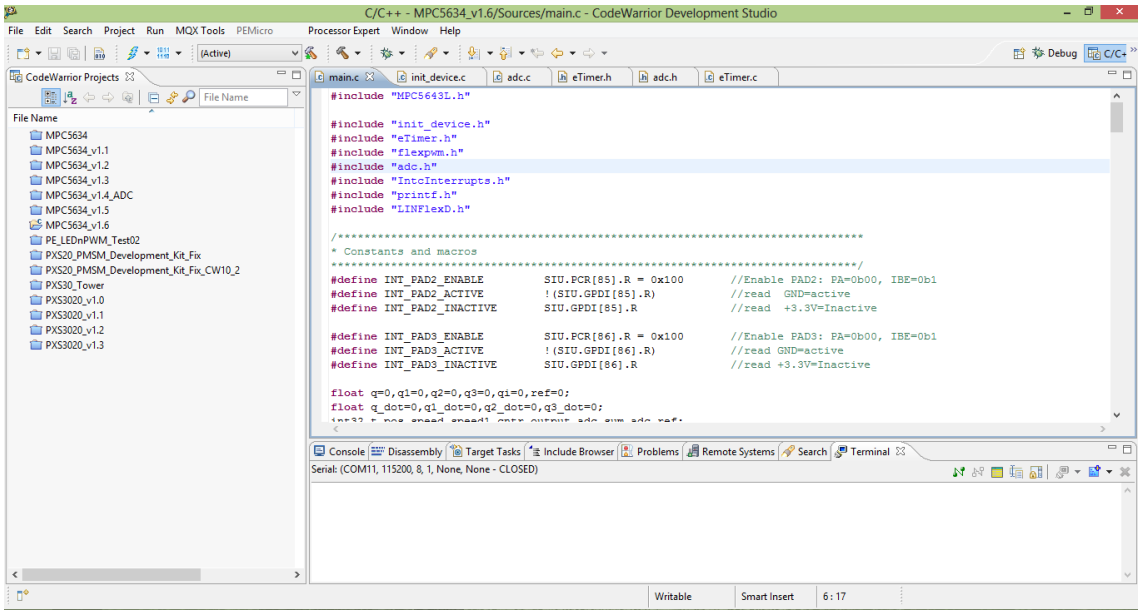


Figura 3.1: CodeWarrior

Sulla sinistra si trova l'elenco dei progetti presenti nel Workspace (cartella dove salva e cerca i progetti CodeWarrior). Al centro si trova l'editor dove viene scritto e modificato il codice. In basso si trovano strumenti come la Console, Dissassembly, Target Task, Include Browser, Problems, Remote systems, Search, e il Terminal. Di questi strumenti i più utilizzati sono il Remote system, per vedere le connessioni con il microcontrollore, il Problems dove vengono descritti, dopo la compilazione, gli errori del codice e il Terminal, dove avvengono le comunicazioni in tempo reale con il microcontrollore. In particolare per utilizzare il terminal bisogna innanzitutto connettere il dispositivo alla porta seriale del computer. Una volta collegato bisognerà selezionare il pulsante selezionato in rosso nella figura 3.2.

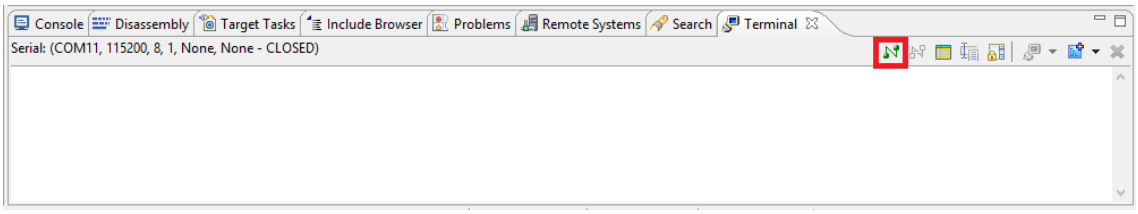


Figura 3.2: Terminal Codewarrior

In seguito si aprirà la finestra di figura 3.3. Qui bisogna impostare le configurazioni della comunicazione tra il terminale e il microcontrollore come il tipo di connessione, la porta utilizzata per il terminale, la velocità di trasmissione e la lunghezza in bit dei dati trasmessi.

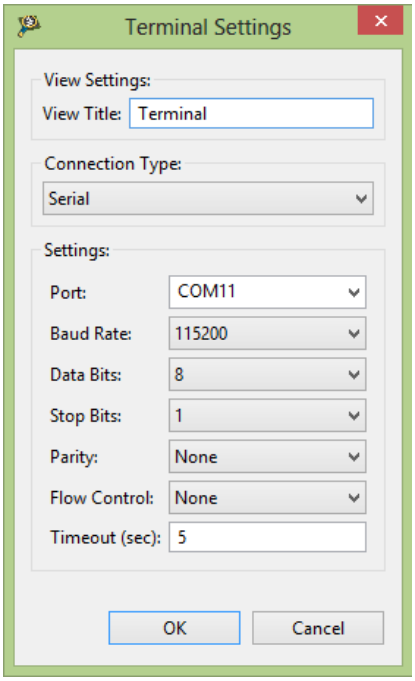


Figura 3.3: Terminal settings

3.1 Creare un nuovo progetto

Per creare un nuovo progetto bisogna aprire la finestra per la creazione Bareboard project che si trova su File>New>Bareboard project. Nel caso in cui il sistema operativo utilizzato fosse linux si può aprire la finestra New Linux/ uClinus Application Project. La finestra di dialogo chiederà inizialmente il nome del progetto e la cartella dove salvarlo. A questo punto chiederà il dispositivo per cui si vuole creare il progetto (figura 3.4).

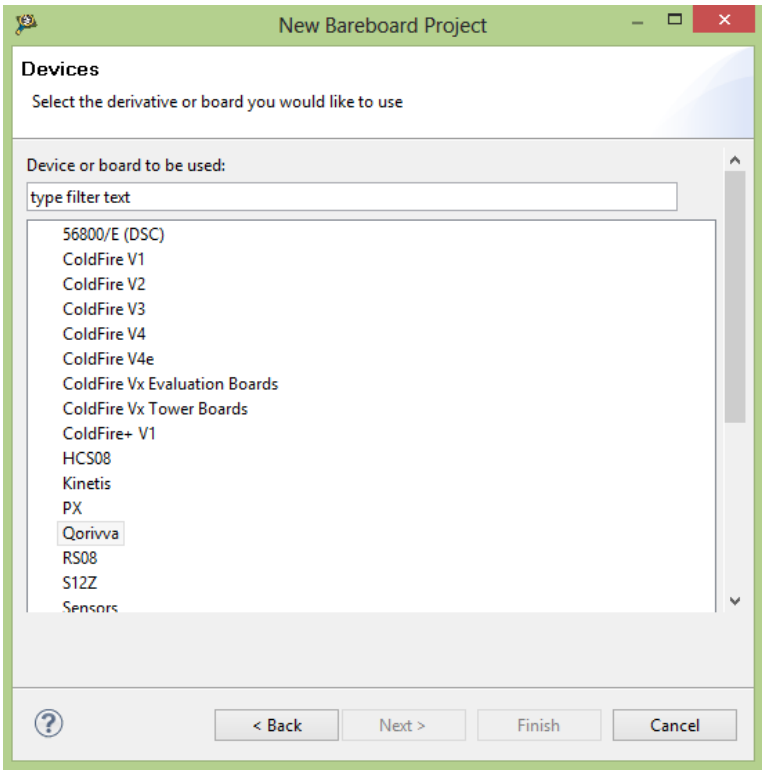


Figura 3.4: scelta dispositivo in CodeWarrior

In seguito viene chiesta la connessione in uso per il progetto e la scelta di modalità di funzionamento tra LSM e DPM (se come dispositivo si è scelti il microcontrollore MPC5643L). Infine viene chiesto il linguaggio di programmazione e se si vuole includere qualche file già esistente. Una volta concluso le impostazioni si preme sul tasto “Finish” e si può iniziare a scrivere il programma.

3.2 Run configuration

Una volta scritto e compilato il codice bisogna caricarlo nel microcontrollore. Per questa operazione si deve selezionare Run>Run. Prima però bisogna configurare la connessione che si trova su Run>Run Configurations.

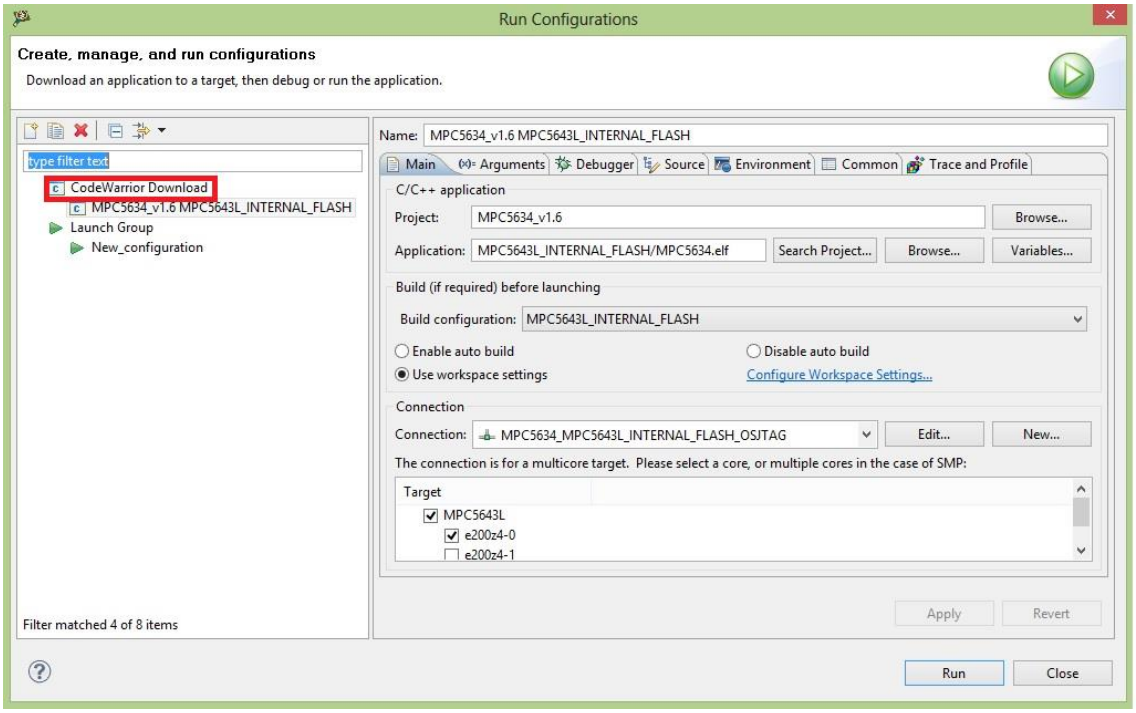


Figura 3.5: finestra di dialogo Run Configuration

Si aprirà la finestra di dialogo come in figura 3.5. Sulla sinistra (riquadro in rosso della figura 3.5) bisogna selezionare, sotto la voce CodeWarrior Download, il progetto che si vuole caricare sul microcontrollore (ci saranno due configurazioni dello stesso progetto ma ciò che cambia è la memoria dove viene caricato).

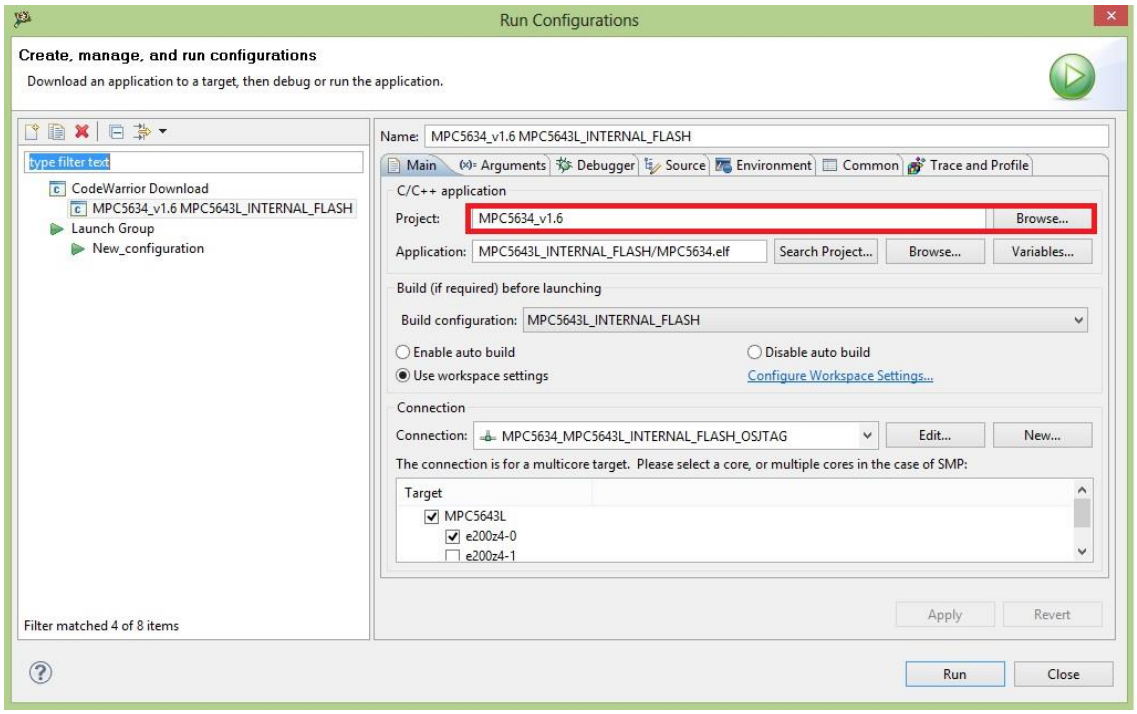


Figura 3.6: scelta progetto da caricare

Una volta aperta la connessione bisogna aprire il progetto che si vuole caricare (premendo il tasto Browse selezionato in rosso nella figura 3.6).

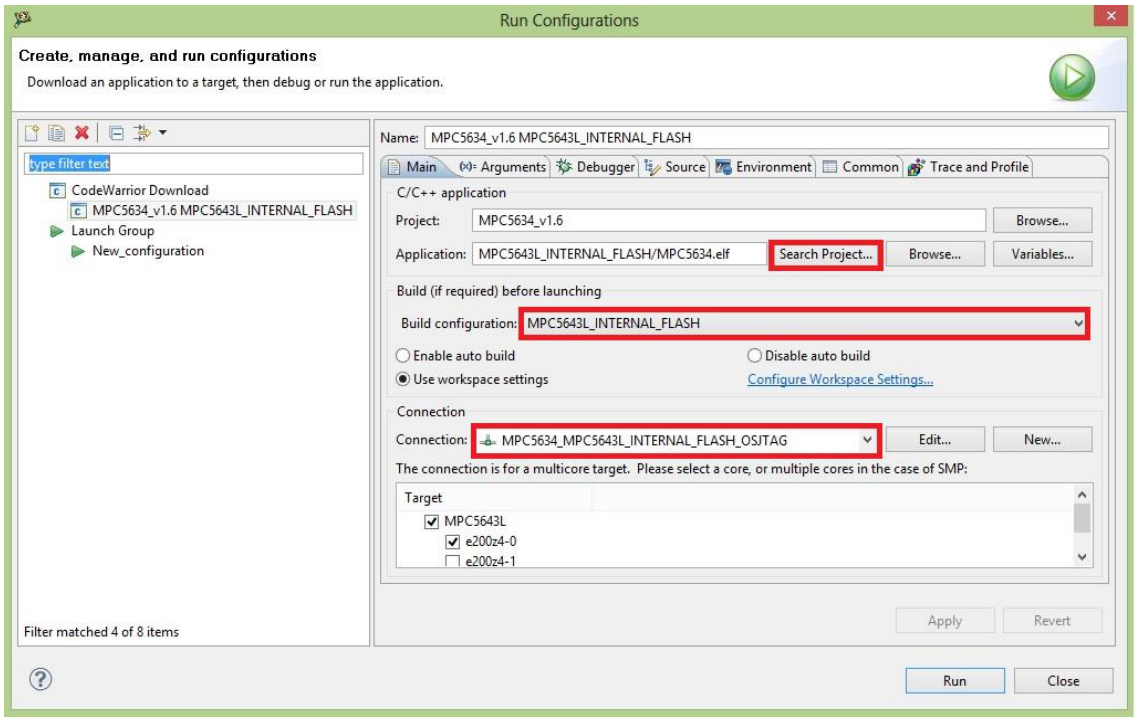


Figura 3.7: scelta memoria in cui caricare il codice

In seguito bisogna andare a scegliere la memoria in cui si vuole caricare il codice nelle tre selezioni segnate nella figura 3.7 in rosso. Come ultima impostazione si deve scegliere i core che si vogliono utilizzare (in caso di microcontrollore multicore).

Una volta configurato la connessione premendo il tasto Run si carica il codice sul microcontrollore.

3.3 Aprire un progetto esistente

Per aprire un progetto esistente si selezionare Import che si trova sul menù File. Si apre una finestra come quella di figura 3.8.

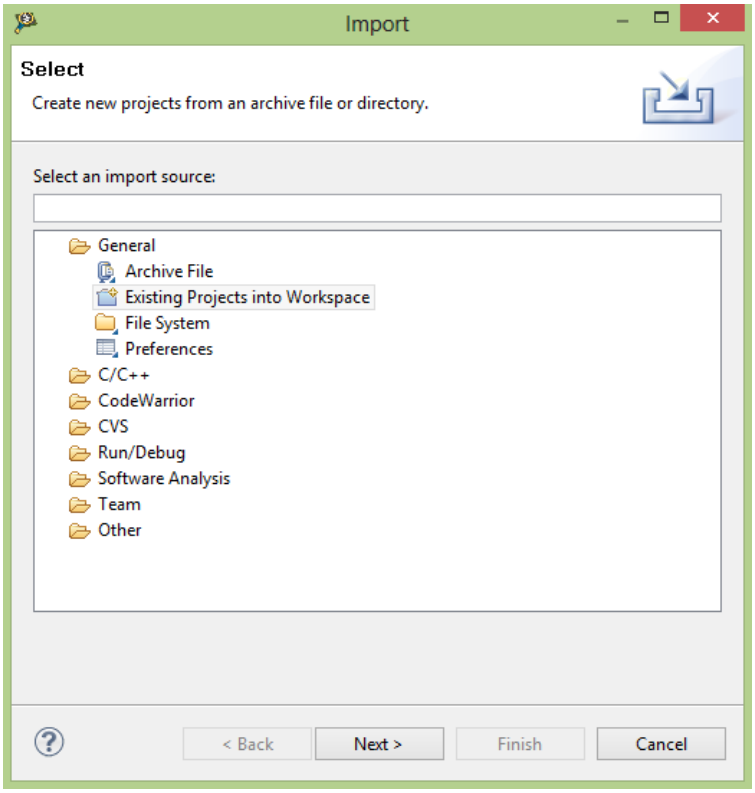


Figura 3.8: finestra Import

Bisogna selezionare “Existing Projects into Workspace” e poi premere Next per passare alla schermata successiva.

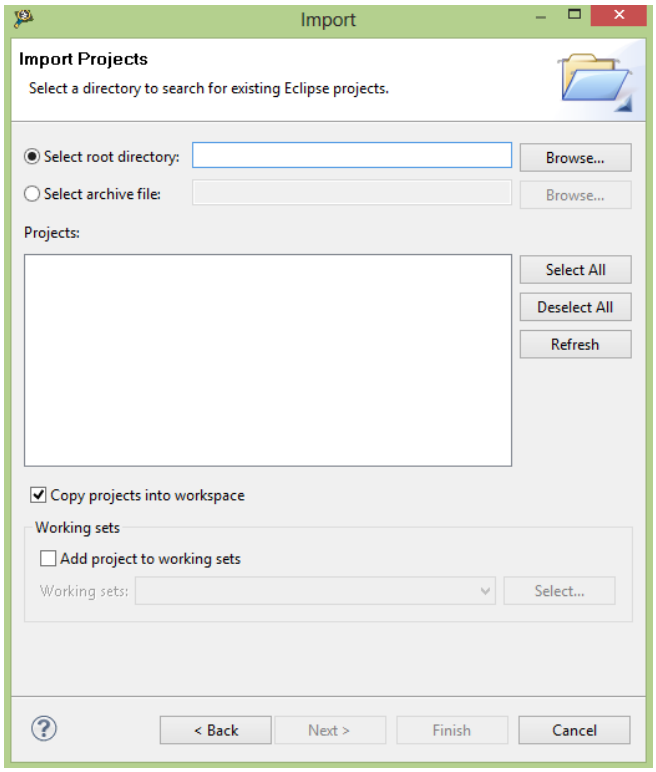


Figura 3.9: scelta progetto schermata Import

Si passerà alla schermata di figura 3.9. Bisogna selezionare la cartella dove si trova il progetto su “Browse” e compariranno i progetti che possono essere aperti. Premendo il tasto “Finish” il progetto verrà aperto e, se è stato selezionato spuntando l’opzione “Copy project into workspace”, il progetto verrà copiato nel workspace dove ci sono tutti i progetti di CodeWarrior.

CAPITOLO 4

Realizzazione progetto

Nel caso di studio preso in considerazione in questo elaborato si ha un microcontrollore che va a comandare un motore controllato in posizione e, per sapere la sua posizione esatta, viene letto un encoder.

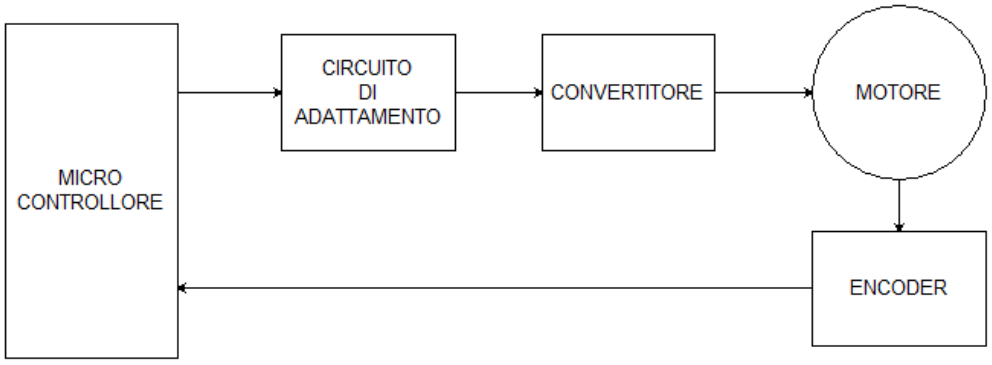


Figura 4.1: schema a blocchi caso di studio

Il microcontrollore dispone solamente di uscite PWM, mentre per comandare il convertitore si deve avere un riferimento di tensione: per questo è stato sono stati inseriti due circuiti elettronici per l'adattamento dell'uscita.

Nel microcontrollore è stato implementato un controllore di tipo PID. Inizialmente si sono trovate le costanti K_P, K_D e K_I del controllore PID analogico e in seguito sono stati discretizzati.

4.1 Circuiti di adattamento

L'uscita PWM proveniente dal microcontrollore varia tra 0V e 3,3V mentre per comandare l'azionamento si deve avere un riferimento di tensione che vari tra -10V e +10V. Per questo motivo è stato inserito un filtro e un circuito zero-span tra i due dispositivi. Il filtro viene inserito per filtrare il segnale di tipo PWM e cercare di avere un riferimento di tensione costante, mentre il circuito zero-span amplifica il segnale portandolo da 0 e 3,3V a 0V e 20V e lo centra sullo zero portandolo tra -10V e +10V.

4.1.2 Filtro

Per questa applicazione è stato scelto un filtro di Butterworth del secondo ordine perché la sua riposta in frequenza nella banda passante è la più piatta possibile (in modulo). Il schema elettrico del circuito è il seguente:

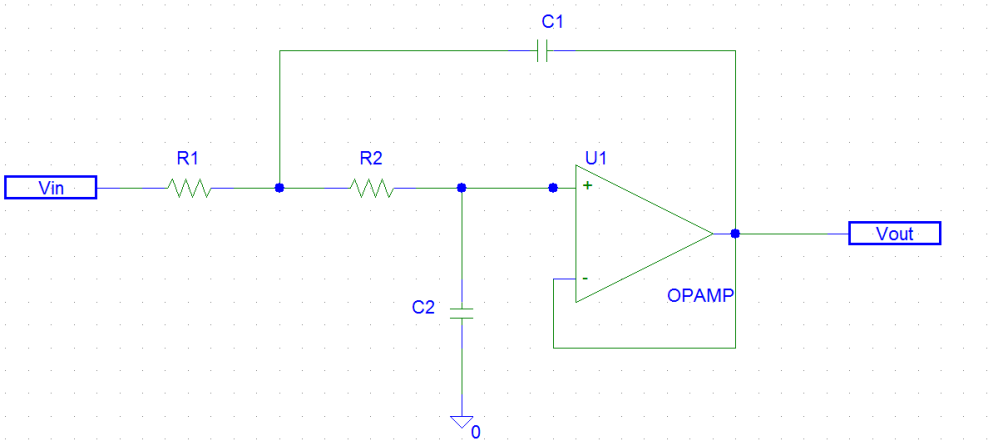


Figura 4.2: filtro di Butterworth

Dal circuito si ricava la seguente funzione di trasferimento:

$$\frac{V_{out}}{V_{in}} = \frac{\frac{1}{R_1 R_2 C_1 C_2}}{s^2 + s\frac{1}{C_1}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) + \frac{1}{R_1 R_2 C_1 C_2}}$$

L'equazione scritta nella forma canonica risulta:

$$\frac{V_{out}}{V_{in}} = \frac{\omega_0^2}{s^2 + s\frac{\omega_0}{Q} + \omega_0^2}$$

Dove

$\omega_0 = \frac{1}{\sqrt{R_1 R_2 C_1 C_2}}$ e $Q = \sqrt{\frac{C_1}{C_2}}\frac{\sqrt{R_1 R_2}}{R_1 + R_2}$

Si è scelto la frequenza di taglio di 240Hz come compromesso tra l'aver un basso ripple del segnale d'uscita ed un adeguato tempo di salita. Il filtro di tipo Butterworth impone che il

fattore di merito Q sia uguale a 1/√2. Infine con questi due dati ci si è calcolato i valori dei singoli componenti.

R1 = R2 = 2,2 KΩ

C1 = 200 nF

C2 = 100 nF

Dal diagramma di Bode del filtro (figura 4.3), con i valori che sono stati calcolati, si osserva la risposta piatta nella banda passante.

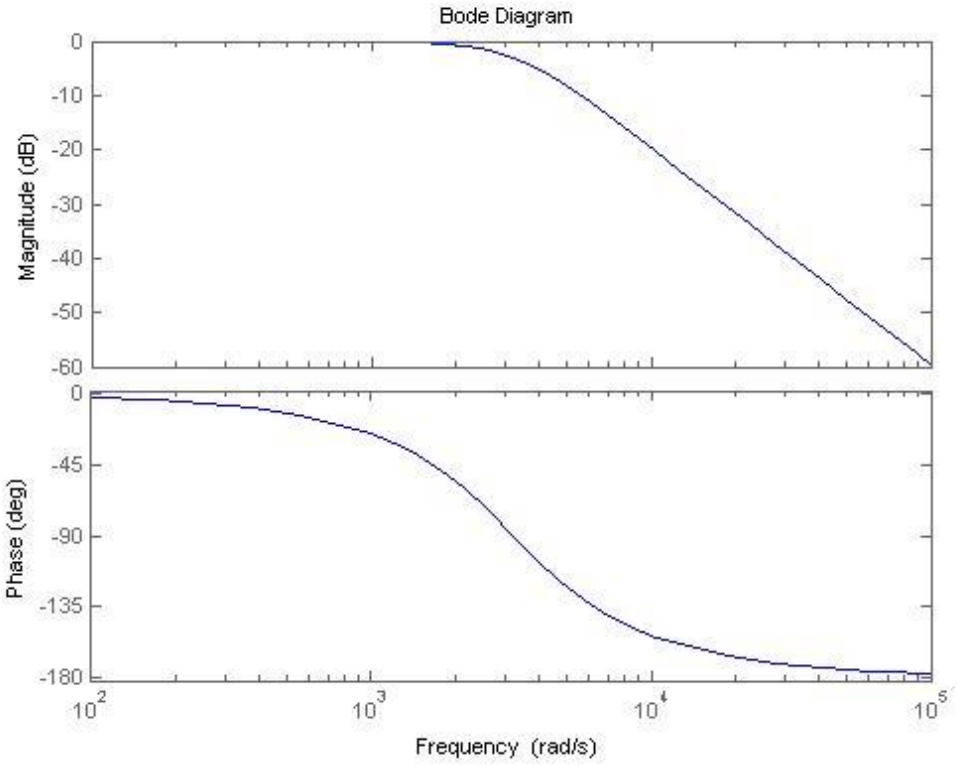


Figura 4.3: diagramma di bode del filtro

4.1.3 Circuito zero-span

Lo schema elettrico del circuito è il seguente:

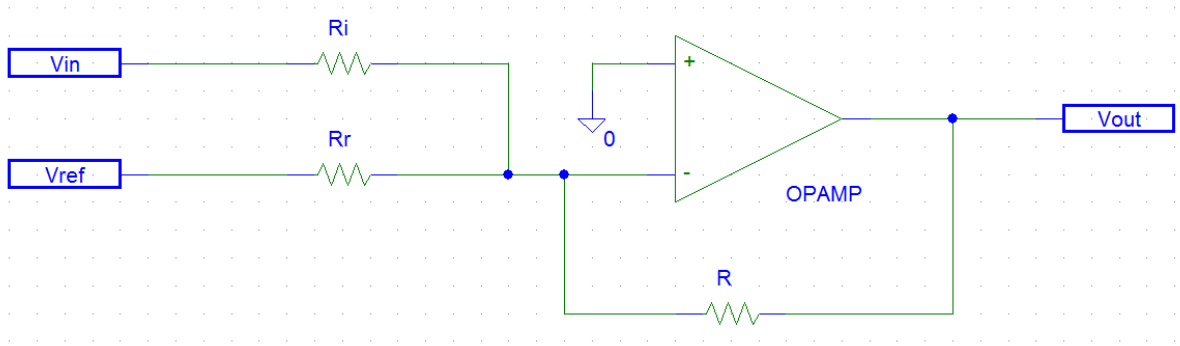


Figura 4.4: circuito zero-span

La formula della funzione di trasferimento è facilmente calcolabile utilizzando la sovrapposizione degli effetti e risulta:

$$V_{out} = -\left(V_{in}\frac{R}{R_i} + V_{ref}\frac{R}{R_r}\right)$$

Dalla formula si vede che il fattore che moltiplica la tensione di ingresso è il guadagno che dovrà avere il circuito, nel nostro caso sarà 20/3,3=6,06. In questo modo la tensione varierà tra 0V e 20V e di conseguenza viene aggiunto un offset per centrare l'uscita sullo zero che rappresenta il secondo termine della funzione di trasferimento. Inoltre si osserva che il circuito è invertente e quindi bisognerà tenerne conto durante l'implementazione software del controllo.

I valori dei componenti sono risultati:

R = 100 KΩ

Rr = 150 KΩ

Ri = 16,5 KΩ

Con una tensione Vref di -15V.

4.2 Controllore digitale PID

Il controllo Proporzionale-Integrale-Derivativo, comunemente abbreviato come PID, è un sistema in retroazione negativa ampiamente impiegato nei sistemi di controllo. Questo controllore produce un segnale di controllo che è la somma di tre termini:

- P proporzionale all'errore
- I proporzionale all'integrale dell'errore
- D proporzionale alla derivata dell'errore

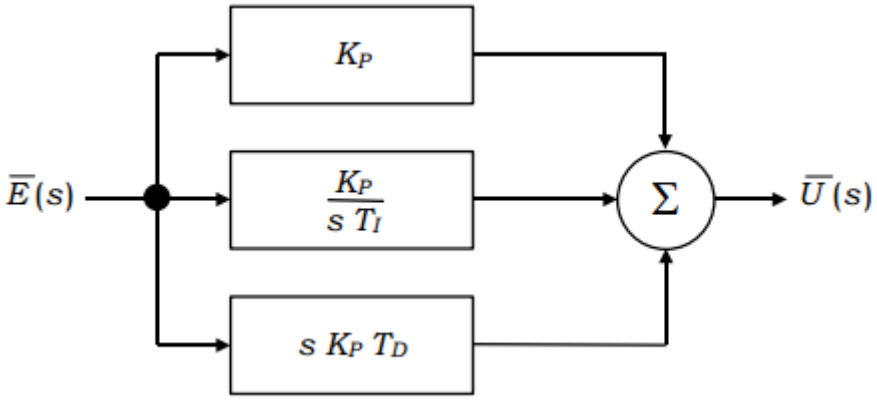


Figura 4.5: schema a blocchi controllore PID

Per i sistemi analogici il controllore PID è caratterizzato dalla seguente funzione di trasferimento:

$$\frac{U(s)}{E(s)} = C(s) = K_P + \frac{K_I}{s} + K_D s = K_P\left[1 + \frac{1}{sT_I} + sT_D\right]$$

Con $T_I = K_P/K_I$ e $T_D = K_D/K_P$.

4.2.1 Calcolo costanti del controllore PID

Il modello del processo da controllare, cioè il motore elettrico, è il seguente:

$$P(s) = \frac{K_{ia}K_{tn}}{B_n + sJ_n}$$

Dove Bn rappresenta il coefficiente di attrito viscoso, Jn l'inerzia del motore, Kia e Ktn le costanti.
Per il calcolo delle costanti si è trascurato Bn, in quanto è molto bassa e non influenza molto il risultato finale. Si sono considerati i seguenti valori del motore:

$J_n = 1.8680 \times 10^{-4}$

$K_{ia} = 2$

$K_{tn} = 0{,}071$

Come caratteristiche che deve soddisfare il controllo di posizione si sono considerate:

Margine di fase $m_\varphi = 60^\circ$

Pulsazione di attraversamento $\omega_c = 2\pi \times 20$ rad/s

Si considera il controllore senza l'integratore, che verrà considerato nel processo.

$$C_{PID}(s) = \frac{K_i}{s}(1 + sT_i + s^2T_iT_d) = \frac{K_i}{s}C^*_{PID}(s)$$

$$P^* = \frac{K_i}{s}\,P(s)$$

Il margine di fase e di guadagno disponibile a ω_c risulta:

$$M = |C^*_{PID}(j\omega_a)| = 20.7734$$

$$\varphi = 1.0472\,rad$$

Considerando T_i=15 x T_d risulta T_i = 0.2112 e T_d = 0.0141, di conseguenza le costanti risultano:

$K_p = 10.3867$

$K_i = 49.1691$

$K_d = 0.1463$

Il controllore PID risulta:

$$C_{PID}(s) = 10{,}3867 + \frac{49{,}1691}{s} + 0{,}1463s$$

4.2.2 Discretizzazione del controllore

L'algoritmo digitale che realizza l'azione PID è ottenuto mediante discretizzazione delle funzioni di trasferimento dei regolatori PID continui. Di seguito viene riportato la discretizzazione utilizzata delle singole azioni del controllore.

Azione integrale

Poiché l'integrale dell'ingresso viene approssimato con una sommatoria, l'uscita del blocco integratore u_i risulta data da:

$$u_i(k) = K_P \frac{T}{T_i} \sum_{0}^{k} e(n)$$

In forma ricorsiva:

$$u_i(k) = u_i(k-1) + K_P \frac{T}{T_i} e(k)$$

Azione derivativa con filtraggio digitale

La realizzazione numerica dell'operazione di derivazione presenta delle difficoltà dovute alla sua elevata sensibilità al rumore sovrapposto al segnale. Per filtrare il segnale si può ricorrere a formule di interpolazione che prendono in considerazione più campioni dell'ingresso.

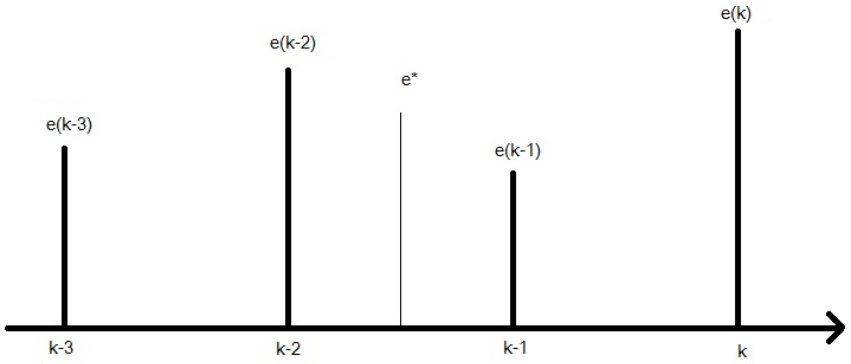


Figura 4.6: filtraggio digitale azione derivativa

Prendendo 4 campioni, la derivata de/dt dell'ingresso e(t) nell'istante kT, viene calcolata come media delle differenze prime negli istanti k-3, k-2, k-1 e k rispetto al valore medio e* dell'ingresso negli istanti considerati, e fittiziamente applicato a metà dell'intervallo kT – (k-3)T.

$$e^* = \frac{1}{4}[e(k) + e(k-1) + e(k-2) + e(k-3)]$$

$$\frac{de}{dt} \cong \frac{1}{4}\left[\frac{e(k) - e^*}{1.5T} + \frac{e(k-1) - e^*}{0.5T} - \frac{e(k-2) - e^*}{0.5T} - \frac{e(k-3) - e^*}{1.5T}\right]$$

Da cui si ricava:

$$\frac{de}{dt} = \frac{1}{6T}[e(k) + 3e(k-1) - 3e(k-2) - e(k-3)]$$

All'aumentare del numero di campioni presi in considerazione si ha una risposta sempre meno pronta ad una improvvisa variazione dell'ingresso e(t), causata ad esempio da un disturbo rapido nel processo. Il termine derivativo risulta quindi:

$$u_d(k) = \frac{K_D}{T}\frac{de(t)}{dt}$$

In conclusione l'algoritmo PID può assumere la seguente forma:

$$u(k) = K_P e(k) + u_i(k) + u_d(k)$$

4.2.3 Implementazione software del controllore

Per il controllo del segnale PWM in uscita bisogna creare una funzione che legga inizialmente il riferimento e l'encoder e, dopo aver calcolato le componenti proporzionale derivativa e integrale dell'errore, le somma e modifica il segnale d'uscita. Alla fine della funzione bisogna salvare i dati in memoria per i calcoli degli istanti successivi. Quest'algoritmo per il controllo dovrà essere eseguito a istanti fissi, per cui si è utilizzato il modulo PIT: viene generato un interrupt ogni millisecondo e viene eseguita la funzione del controllo PID. Inoltre è stata fatta una limitazione sull'uscita in quanto il motore può ricevere al massimo 3A. Di seguito viene riportato il codice, semplificato, della funzione di controllo con la discretizzazione riportata nel paragrafo precedente.

```
static void CONTROL_OUT(void)
{
        // Data Read
        cont     = value_eTimer();
        adc_ref = value_ADC();

        //Calcolate data
        q        = 2*3.14*((cont-adc_ref))/MAX_VAL;
        q_dot    = (1/(6*T)*(q +3*q1-3*q2-q3);

        //Control Update
        qi = q*0.001+qi;              //Store value for integral
      //Calcolate output PID
      cntr_output = Kp*q+Kd*q_dot+Ki*qi;

      //Limit output to motor to 3A
      if(cntr_output>=307)
        {
              cntr_output=307;
        }else if (cntr_output<=-307)
              cntr_output=-307;

        //Change output
        flexpwm0_sub0_update_val3(cntr_output);

        // Update data
        q3      = q2;
        q3_dot  = q2_dot;
        q2      = q1;
        q2_dot  = q1_dot;
        q1      = q;
        q1_dot  = q_dot;

        //Clear flag interrupt
        PIT.TFLG0.B.TIF = 1;
}
```

4.3 Risultati sperimentali

Per visualizzare il corretto funzionamento del controllore di posizione si ha imposto il riferimento a π/2 rad e, facendo partire il motore da 0 rad, si sono misurate le posizioni che assumeva il motore. Le misurazioni sono state raccolte ogni 20ms attraverso la porta seriale della Tower system.

Figura 4.7: posizioni assunte dal motore con riferimento di π/2 rad

La figura 4.7 mostra il risultato delle posizioni del motore con riferimento di π/2 rad. Il controllore PID era stato calcolato anche con pulsazioni di attraversamento ω_C = 10x2π rad e ω_C = 30x2π rad. È stato effettuato la stessa misurazione e i risultati sono visibili nelle figure 4.8 e 4.9.

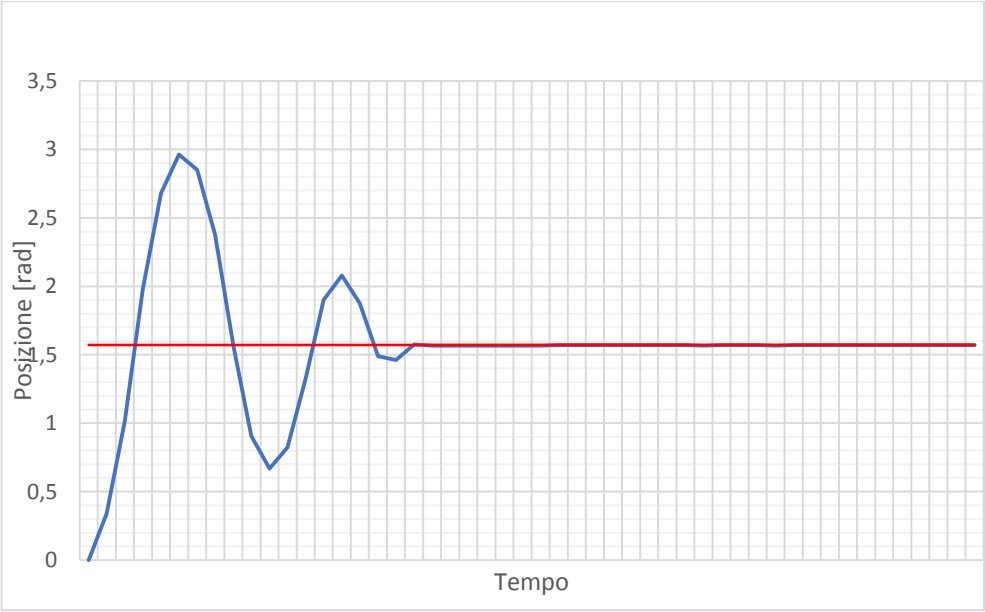


Figura 4.8: posizioni assunte dal motore con riferimento di π/2 rad e pulsazione di attraversamento di ω_C = 30x2π rad

Figura 4.8: posizioni assunte dal motore con riferimento di π/2 rad e pulsazione di attraversamento di ω_C = 10x2π rad

Come si vede dai grafici la scelta della pulsazione di attraversamento è stata una soluzione di compromesso tra l'avere uno smorzamento contenuto e un tempo di assestamento accettabile.

CONCLUSIONI

In questo elaborato si è voluto offrire uno strumento di consultazione per le persone che vogliono iniziare a programmare un microcontrollore della Freescale per la prima volta. L'obbiettivo era quello di illustrare le caratteristiche tecniche e le potenzialità di un sistema di controllo basato su un microcontrollore e di insegnare le basi per l'uso del sistema di sviluppo CodeWarrior. Inoltre, implementando il codice per il caso di studio sviluppato in questo elaborato, si è voluto fornire un esempio chiaro di come si gestiscano i vari moduli del microcontrollore e come realizzare un controllo di posizione di un motore elettrico.

BIBLIOGRAFIA

T. N. Blalock and R. C. Jaeger, Microelettronica, McGraw-Hill, 3rd ed., 2009.

Nicola Bogoni. Progetto, analisi dei rischi e realizzazione di dispositivi aptici per la riabilitazione. Tesi di laurea, Università degli studi di Padova, Facoltà di ingegneria, 2012.

Prof. Doriano Ciscato. “Appunti di controllo digitale”.

Freescale Semiconductor. Qorivva MPC5643L Microcontroller Reference Manual Rev. 9, Mar 2012.

Freescale Semiconductor CodeWarrior Development Studio for Microcontrollers V10.x Targeting Manual, April 2013.

APPENDICE

A. Listati di codice

Il progetto è diviso in più file dove ognuno si occupa di un modulo specifico del microcontrollore. Sono stati quindi implementati i file eTimer.c, flexPWM.c, adc.c e init_device.c, con i relativi file header, e il main dove è stato implementato il controllo PID.

A.1 Istruzioni in linguaggio C per leggere l'encoder

Il seguente codice inizializza il canale 0 e il canale 1 del modulo eTimer 0. Il canale 0 deve contare da -1000 a +1000. Quando il contatore arriva a 1000 il CMPLD1 lo re-inizializza a -1000, mentre se arriva a -1000 il CMPLD2 lo re-inizializza a 1000. Si è utilizzato come modalità di conteggio la quadrature count mode. Il canale 1 deve contare da 0 a 1 e conta solamente i fronti di salita. Il canale 1 inoltre è impostato come Master: in questo modo quando l'encoder avrà fatto un giro, nel contatore del canale 0 viene caricato il LOAD register, che è impostato a 0. Per fare l'azzeramento ogni volta che l'encoder ha fatto un giro il campo OUTMODE del registro CTRL2 del canale 1 è stato impostato con "set on successful compare on COMP1 cleared on successful compare on COMP2", cioè la flag responsabile della reinizializzazione del canale 0 viene posta a 1 quando il contatore del canale 1 risulta uguale a COMP1 (posto uguale a 1) e viene cancellata quando il contatore risulta uguale a COMP2 (posto uguale a 0).
Di seguito è riportato il file header "eTimer.h". Per una più facile lettura e modifica dei registri, nel file header sono state create delle macro con i valori che assumono i registri nelle varie configurazioni. Inoltre l'header contiene alla fine i prototipi delle funzioni specificate nel file "eTimer.c".

```
#ifndef ETIMER_H_
#define ETIMER_H_

#include "MPC5643L.h"

/*****************************************************************
* eTimer MACRO definitions
*****************************************************************/

#define eTimer0_COMP1_CH0              0x03E8
#define eTimer0_COMP2_CH0              0xFC18
#define eTimer0_CMPLD1_CH0             0xFC18
#define eTimer0_CMPLD2_CH0             0x03E8
#define eTimer0_LOAD_CH0               0x0000

#define eTimer0_COMP1_CH1              0x0001
#define eTimer0_COMP2_CH1              0x0000
#define eTimer0_CMPLD1_CH1             0x0000
#define eTimer0_CMPLD2_CH1             0x0000
```

```
#define eTimer0_LOAD_CH1                0x0000

#define eTimer_ENBL                     0x003F

// Count mode control register 1
#define NO_OPERATIONS                                                   0x0
#define COUNT_RISING_EDGES_OF_PRIMARY_SRC                               0x1
#define COUNT_RISING_AND_FALLING_EDGES_OF_PRIMARY_SRC                   0x2
#define COUNT_RISING_EDGES_OF_PRIMSRC_WHILE_SECSRC_HIGH                 0x3
#define QUADRATURE_COUNT_MODE                                           0x4
#define COUNT_RISING_EDGES_OF_PRIMSRC_SECSRC_SPECIFIES_DIR              0x5
#define EDGE_OF_SECSRC_TRIG_PRIMARY_COUNT_TILL_CMP                      0x6
#define CASCADED_COUNT_MODE                                             0x7

// primary, secondary count sources
#define COUNTER_0_INPUT                                                 0x0
#define COUNTER_1_INPUT                                                 0x1
#define COUNTER_2_INPUT                                                 0x2
#define COUNTER_3_INPUT                                                 0x3
#define COUNTER_4_INPUT                                                 0x4
#define COUNTER_5_INPUT                                                 0x5
#define AUX_INPUT_0                                                     0x8
#define AUX_INPUT_1                                                     0x9
#define AUX_INPUT_2                                                     0xa
#define COUNTER_0_OUTPUT                                                0x10
#define COUNTER_1_OUTPUT                                                0x11
#define COUNTER_2_OUTPUT                                                0x12
#define COUNTER_3_OUTPUT                                                0x13
#define COUNTER_4_OUTPUT                                                0x14
#define COUNTER_5_OUTPUT                                                0x15
#define IPBCLK_DIV_BY_1                                                 0x18
#define IPBCLK_DIV_BY_2                                                 0x19
#define IPBCLK_DIV_BY_4                                                 0x1A
#define IPBCLK_DIV_BY_8                                                 0x1B
#define IPBCLK_DIV_BY_16                                                0x1C
#define IPBCLK_DIV_BY_32                                                0x1D
#define IPBCLK_DIV_BY_64                                                0x1E
#define IPBCLK_DIV_BY_128                                               0x1F

// count duration, one shot or continuous
#define COUNT_UNTIL_COMPARE_AND_STOP                                    0x1
#define COUNT_REPEATEDLY                                                0x0

// count lenght
#define COUNT_UNTIL_COMPARE_AND_REINITIALIZE                            0x1
#define COUNT_TO_ROLL_OVER                                              0x0

// count direction
#define COUNT_UP                                                        0x0
#define COUNT_DOWN                                                      0x1

// Output Enable OEN
#define OFLAG_DISABLE                                                   0x0
#define OFLAG_ENABLE                                                    0x1

// Redundant Channel RDNT
#define REDUNDANT_CHANNEL_CHECKING_DISABLE                              0x0
#define REDUNDANT_CHANNEL_CHECKING_ENABLE                               0x1

//Co-channel Initialization
#define NO_REINIZIALIZATION                                             0x0
```

```
#define REINIZIALIZATION_LOAD                                   0x1
#define REINIZIALIZATION_CMPLD                                  0x2

// Polarity selection
#define INVERTED_POLARITY                                       0x1
#define TRUE_POLARITY                                           0x0

// Master Mode MSTR
#define ENABLE_BROADCAST_OF_CMP_EVENTS                          0x1
#define DISABLE_BROADCAST_OF_CMP_EVENTS                         0x0

// Output Mode OUTMODE
#define SOFTWARE_CONTROLLED                                     0x0
#define CLEAR_ON_COMPARE                                        0x1
#define SET_ON_COMPARE                                          0x2
#define TOGGLE_ON_COMPARE                                       0x3
#define TOGGLE_USING_ALTERNATING_COMP_REGISTERS                 0x4
#define SET_ON_COMP1_COMPARE_CLEAR_ON_SECSRC_EDGE               0x5
#define SET_ON_COMP2_COMPARE_CLEAR_ON_SECSRC_EDGE               0x6
#define SET_ON_COMPARE_CLEAR_ON_ROLL_OVER                       0x7
#define SET_ON_COMP1_COMPARE_CLEAR_ON_COMP2_COMPARE             0x8
#define ASSERT_WHILE_CNTR_ACTIVE_CLEAR_WHEN_CNTR_STOPPED        0x9
#define ASSERT_WHEN_COUNTING_UP_CLEAR_WHEN_COUNTING_DOWN        0xa
#define ENABLE_GATED_CLOCK_OUTPUT_WHILE_CNTR_ACTIVE             0xf

// Stop Actions Enable
#define OUTPUT_ENABLE_STOP_MODE                                 0x0
#define OUTPUT_ENABLE_DISABLED_DURING_STOP_MODE                 0x1

// Reload on Capture
#define NOT_RELOAD_ON_CAPTURE_EVENT                              0x0
#define RELOAD_ON_CAPTURE1_EVENT                                 0x1
#define RELOAD_ON_CAPTURE2_EVENT                                 0x2
#define RELOAD_ON_CAPTURE1_AND_CAPTURE2_EVENT                    0x3

// Debug Actions Enable
#define NORMAL_OPERATION_DEBUG_MODE                              0x0
#define HALT_CHANNEL_COUNTER_DEBUG_MODE                          0x1
#define FORCE_OFLAG_DEBUG_MODE                                   0x2
#define HALT_CHANNEL_COUNTER_AND_FORCE_OFLAG_DEBUG_MODE          0x3

// Compare Load Control
#define NEVER_PRELOAD                                            0x0
#define LOAD_COMP_WITH_CMPLD1_UPON_SUCCESSFUL_COMPARE_WITH_COMP1 0x2
#define LOAD_COMP_WITH_CMPLD1_UPON_SUCCESSFUL_COMPARE_WITH_COMP2 0x3
#define LOAD_COMP_WITH_CMPLD2_UPON_SUCCESSFUL_COMPARE_WITH_COMP1 0x4
#define LOAD_COMP_WITH_CMPLD2_UPON_SUCCESSFUL_COMPARE_WITH_COMP2 0x5
#define LOAD_CNTR_WITH_CMPLD_UPON_SUCCESSFUL_COMPARE_WITH_COMP1  0x6
#define LOAD_CNTR_WITH_CMPLD_UPON_SUCCESSFUL_COMPARE_WITH_COMP2  0x7

// Compare Mode
#define COMP1_UP_COMP2_UP                                        0x0
#define COMP1_DOWN_COMP2_UP                                      0x1
#define COMP1_UP_COMP2_DOWN                                      0x2
#define COMP1_DOWN_COMP2_DOWN                                    0x3

// Capture Mode Control
#define DISABLE                                                  0x0
#define FALLING_EDGE                                             0x1
#define RISING_EDGE                                              0x2
#define ANY_EDGE                                                 0x3
```

```
// One-Shot Capture Mode
#define FREE_RUNNIG                                                    0x0
#define ONE_SHOT                                                       0x1

// Arm Capture
#define DISABLE_INPUT_CAPTURE                                          0x0
#define ENABLE_INPUT_CAPTURE                                           0x1


/*********************************************************************
* Functions
*********************************************************************/
void etimer0_init(void);
uint16_t value_eTimer(void);


#endif ETIMER_H /*ETIMER_H_ */
```

Di seguito viene riportato il file “eTimer.c” che contiene le funzioni etimer0_init() e value_eTimer(). La funzione etimer0_init() inizializza i registri del eTimer 0 canale 0 e canale 1 con le impostaziono riportate sopra, mentre value_eTimer() è una semplice funzione che restituisce il valore del contatore che sarà utile durante il controllo del motore.

```
void etimer0_init(void)
{

      /*---------------------------------------------------------------
     * eTimer #0 Control registers
     ---------------------------------------------------------------*/

      /* ENBL register: enable the eTimer 0 */
      ETIMER_0.ENBL.R= eTimer_ENBL;

      /*---------------------------------------------------------------
       * eTimer #0 CHANNEL #0 Control registers
       ---------------------------------------------------------------*/
      /* COMP1 register */
      ETIMER_0.CHANNEL[0].COMP1.R= eTimer0_COMP1_CH0;

      /* COMP2 register */
      ETIMER_0.CHANNEL[0].COMP2.R= eTimer0_COMP2_CH0;

      /* LOAD register */
      ETIMER_0.CHANNEL[0].LOAD.R= eTimer0_LOAD_CH0;

      /* CTRL register: Quadrature count mode, primary count source 0
      input pin, Count repeatedly,
      Continue counting to roll over, Count Direction up, secondary
      count source 1 input pin  */
      ETIMER_0.CHANNEL[0].CTRL.B.CNTMODE= QUADRATURE_COUNT_MODE;
      ETIMER_0.CHANNEL[0].CTRL.B.PRISRC= COUNTER_0_INPUT;
      ETIMER_0.CHANNEL[0].CTRL.B.ONCE= COUNT_REPEATEDLY;
      ETIMER_0.CHANNEL[0].CTRL.B.LENGTH=
```

```
                  COUNT_UNTIL_COMPARE_AND_REINITIALIZE;
ETIMER_0.CHANNEL[0].CTRL.B.DIR= COUNT_UP;
ETIMER_0.CHANNEL[0].CTRL.B.SECSRC= COUNTER_1_INPUT;

/* CTRL2 register: Other channels may force a re-initialization
of this channel's counter using the LOAD reg */
ETIMER_0.CHANNEL[0].CTRL2.B.OEN= OFLAG_DISABLE;
ETIMER_0.CHANNEL[0].CTRL2.B.RDNT=
                  REDUNDANT_CHANNEL_CHECKING_DISABLE;
ETIMER_0.CHANNEL[0].CTRL2.B.COINIT= REINIZIALIZATION_LOAD;
ETIMER_0.CHANNEL[0].CTRL2.B.SIPS= TRUE_POLARITY;
ETIMER_0.CHANNEL[0].CTRL2.B.PIPS= TRUE_POLARITY;
ETIMER_0.CHANNEL[0].CTRL2.B.OPS= TRUE_POLARITY;
ETIMER_0.CHANNEL[0].CTRL2.B.MSTR=
                  DISABLE_BROADCAST_OF_CMP_EVENTS;
ETIMER_0.CHANNEL[0].CTRL2.B.OUTMODE= SOFTWARE_CONTROLLED;

/* CTRL3 register: Output enable is unaffected by stop mode */
ETIMER_0.CHANNEL[0].CTRL3.B.STPEN= OUTPUT_ENABLE_STOP_MODE;
ETIMER_0.CHANNEL[0].CTRL3.B.ROC= NOT_RELOAD_ON_CAPTURE_EVENT;
ETIMER_0.CHANNEL[0].CTRL3.B.DBGEN=NORMAL_OPERATION_DEBUG_MODE;

/* CMPLD1 register */
ETIMER_0.CHANNEL[0].CMPLD1.R= eTimer0_CMPLD1_CH0;

/* CMPLD2 register */
ETIMER_0.CHANNEL[0].CMPLD2.R= eTimer0_CMPLD2_CH0;

/* CCCTRL register: Compare and Capture are not used, the
register are reset */
ETIMER_0.CHANNEL[0].CCCTRL.B.CLC2=
      LOAD_CNTR_WITH_CMPLD_UPON_SUCCESSFUL_COMPARE_WITH_COMP2;
ETIMER_0.CHANNEL[0].CCCTRL.B.CLC1=
      LOAD_CNTR_WITH_CMPLD_UPON_SUCCESSFUL_COMPARE_WITH_COMP1;
ETIMER_0.CHANNEL[0].CCCTRL.B.CMPMODE= COMP1_UP_COMP2_DOWN;
ETIMER_0.CHANNEL[0].CCCTRL.B.CPT2MODE= DISABLE;
ETIMER_0.CHANNEL[0].CCCTRL.B.CPT1MODE= DISABLE;
ETIMER_0.CHANNEL[0].CCCTRL.B.ONESHOT= FREE_RUNNIG;
ETIMER_0.CHANNEL[0].CCCTRL.B.ARM= DISABLE_INPUT_CAPTURE;

/*-----------------------------------------------------------------
 * eTimer #0 CHANNEL #1 Control registers
 -----------------------------------------------------------------*/

/* COMP1 register */
ETIMER_0.CHANNEL[1].COMP1.R= eTimer0_COMP1_CH1;

/* COMP2 register */
ETIMER_0.CHANNEL[1].COMP2.R= eTimer0_COMP2_CH1;

/* LOAD register */
ETIMER_0.CHANNEL[1].LOAD.R= eTimer0_LOAD_CH1;

/* CTRL register : Count rising edges of primary source, primary
count source 2 input pin, Count repeatedly,
Count until compare then reinitialize, count up */
ETIMER_0.CHANNEL[1].CTRL.B.CNTMODE=
                  COUNT_RISING_EDGES_OF_PRIMARY_SRC;
ETIMER_0.CHANNEL[1].CTRL.B.PRISRC= COUNTER_4_INPUT;
ETIMER_0.CHANNEL[1].CTRL.B.ONCE= COUNT_REPEATEDLY;
ETIMER_0.CHANNEL[1].CTRL.B.LENGTH=
```

```
                        COUNT_UNTIL_COMPARE_AND_REINITIALIZE;
      ETIMER_0.CHANNEL[1].CTRL.B.DIR= COUNT_UP;
      ETIMER_0.CHANNEL[1].CTRL.B.SECSRC= COUNTER_4_INPUT;

      /* CTRL2 register: Master Mode, Set on successful compare on
      COMP1 clear on successful compare on COMP2  */
      ETIMER_0.CHANNEL[1].CTRL2.B.OEN= OFLAG_DISABLE;
      ETIMER_0.CHANNEL[1].CTRL2.B.RDNT=
                        REDUNDANT_CHANNEL_CHECKING_DISABLE;
      ETIMER_0.CHANNEL[1].CTRL2.B.COINIT= NO_REINIZIALIZATION;
      ETIMER_0.CHANNEL[1].CTRL2.B.SIPS= TRUE_POLARITY;
      ETIMER_0.CHANNEL[1].CTRL2.B.PIPS= TRUE_POLARITY;
      ETIMER_0.CHANNEL[1].CTRL2.B.OPS= TRUE_POLARITY;
      ETIMER_0.CHANNEL[1].CTRL2.B.MSTR=ENABLE_BROADCAST_OF_CMP_EVENTS;
      ETIMER_0.CHANNEL[1].CTRL2.B.OUTMODE=
                        SET_ON_COMP1_COMPARE_CLEAR_ON_COMP2_COMPARE;

      /* CTRL3 register: Output enable is unaffected by stop mode */
      ETIMER_0.CHANNEL[1].CTRL3.B.STPEN= OUTPUT_ENABLE_STOP_MODE;
      ETIMER_0.CHANNEL[1].CTRL3.B.ROC= NOT_RELOAD_ON_CAPTURE_EVENT;
      ETIMER_0.CHANNEL[1].CTRL3.B.DBGEN= NORMAL_OPERATION_DEBUG_MODE;

      /* CMPLD1 register */
      ETIMER_0.CHANNEL[1].CMPLD1.R= eTimer0_CMPLD1_CH1;

      /* CMPLD2 register */
      ETIMER_0.CHANNEL[1].CMPLD2.R= eTimer0_CMPLD2_CH1;

      /* CCCTRL register: Compare and Capture are not used, the
      register are reset */
      ETIMER_0.CHANNEL[1].CCCTRL.B.CLC2=
            LOAD_CNTR_WITH_CMPLD_UPON_SUCCESSFUL_COMPARE_WITH_COMP2;
      ETIMER_0.CHANNEL[1].CCCTRL.B.CLC1=
            LOAD_CNTR_WITH_CMPLD_UPON_SUCCESSFUL_COMPARE_WITH_COMP1;
      ETIMER_0.CHANNEL[1].CCCTRL.B.CMPMODE= COMP1_UP_COMP2_UP;
      ETIMER_0.CHANNEL[1].CCCTRL.B.CPT2MODE= DISABLE;
      ETIMER_0.CHANNEL[1].CCCTRL.B.CPT1MODE= DISABLE;
      ETIMER_0.CHANNEL[1].CCCTRL.B.ONESHOT= FREE_RUNNIG;
      ETIMER_0.CHANNEL[1].CCCTRL.B.ARM= DISABLE_INPUT_CAPTURE;

      /*STS register: clear flag IEHF*/
      ETIMER_0.CHANNEL[1].STS.B.IEHF = 1;

      /*INTDMA register: Enable interrupt from etimer*/
      ETIMER_0.CHANNEL[1].INTDMA.B.IEHFIE=1;
}

uint16_t value_eTimer(void)
{
      uint16_t val;
      val = ETIMER_0.CHANNEL[0].CNTR.R;
      return val;
}
```

A.3 Istruzioni in linguaggio C per la generazione del segnale PWM

Nel seguente codice viene generato un segnale PWM dal modulo flexPWM_0 submodulo 0. Il contatore è stato impostato per funzionare fino a 12 bit (conta da -2048 a 2048), per questo è stato impostato INIT=0xF800 e VAL1=0x0800. Il segnale è allineato al fronte quindi i valori di VAL2 e VAL4 sono stati impostati uguali a INIT. Il clock utilizzato è quello dell'IP Bus senza prescaler. I due segnali sono indipendenti e non si è utilizzato il deadtime. Nel file header "flexpwm.h" sono state create come nel caso dell'eTimer una serie di macro che servono per capire e modificare più velocemente il codice e poi sono presenti i prototipi delle funzioni.

```
#ifndef _FLEXPWM_H_
#define _FLEXPWM_H_

#include "MPC5643L.h"

/*****************************************************************
* flexPWM MACRO definitions
*****************************************************************/
#define FlexPWM0_INIT_sub0              0xF800
#define FlexPWM0_VAL_sub0               0x0000
#define FlexPWM0_VAL1_sub0              0x0800
#define FlexPWM0_VAL2_sub0              0xF800
#define FlexPWM0_VAL3_sub0              0x0000
#define FlexPWM0_VAL4_sub0              0xF800
#define FlexPWM0_VAL5_sub0              0x0000
#define FlexPWM0_DISMAP_sub0            0x0000
#define FlexPWM0_DTCNT0_sub0            0x0FFF
#define FlexPWM0_DTCNT1_sub0            0x0FFF

// PWM Output Enables
#define PWM_ENABLE                                      0xF
#define PWM_DISABLE                                     0x0

// PWM Masks
#define OUTPUT_NORMAL                                   0x0
#define OUTPUT_MASKED                                   0xF

// Software Controlled Output
#define LOGIC_0_DEADTIME_GENERATOR                      0x0
#define LOGIC_1_DEADTIME_GENERATOR                      0x1

// PWM Control Select
#define GENERATED_PWM_USED_DEADTIME                     0x0
#define INVERTED_PWM_USED_DEADTIME                      0x1
#define OUT_USED_DEATIME                                0x2

// Fault Level
#define LOGIC_0_FAULT                                   0x0
#define LOGIC_1_FAULT                                   0xF

// Automatic Fault Clearing
#define MANUAL_FAULT_CLEARING                           0x0
#define AUTOMATIC_FAULT_CLEARING                        0xF

// Fault Safety Mode
```

```
#define NORMAL_MODE                                    0x0
#define SAFETY_MODE                                    0xF

// Fault Interrupt Enables
#define FAULT_INTERRUPT_ENABLE                         0xF
#define FAULT_INTERRUPT_DISABLE                        0x0

// Fault Glitch Stretch Enable
#define FAULT_GLITCH_STRETCHING_DISABLE                0x0000
#define FAULT_GLITCH_STRETCHING_ENABLE                 0x8000

// Fault Filter Count
#define Fault_Filter_Count                             0x0

// Fault Filter Period
#define Fault_Filter_Period                            0x1

// Independent or Complementary Pair Operation
#define COMPLEMENTARY_PWM                              0x0
#define INDIPENDENT_PWM                                0x1

// Initialization Control Select
#define INITIALIZATION_PWMX                            0x0
#define INITIALIZATION_MASTER_RELOAD                   0x1
#define INITIALIZATION_MASTER_SYNC                     0x2
#define INITIALIZATION_EXT_SYNC                        0x3

// Force Initialization Enable
#define INITIALIZATION_FORCE_OUT_DISABLE               0x0
#define INITIALIZATION_FORCE_OUT_ENABLE                0x1

// Force Source Select
#define LOCAL_FORCE_SIGNAL                             0x0
#define MASTER_FORCE_SIGNAL                            0x1
#define LOCAL_RELOAD_SIGNAL                            0x2
#define MASTER_RELOAD_SIGNAL                           0x3
#define LOCAL_SYNC_SIGNAL                              0x4
#define MASTER_SYNC_SIGNAL                             0x5
#define EXTERNAL_FORCE_SIGNAL                          0x6

// Reload Source Select
#define LOCAL_RELOAD_USED_TO_RELOAD_REGISTERS          0x0
#define MASTER_RELOAD_USED_TO_RELOAD_REGISTERS         0x1

// Clock Source Select
#define IP_BUS_CLOCK                                   0x0
#define EXT_CLOCK                                      0x1
#define AUX_CLK                                        0x2

// Half Cycle Reload
#define HALF_CYCLE_RELOAD_DISABLE                      0x0
#define HALF_CYCLE_RELOAD_ENABLE                       0x1

// Full Cycle Reload
#define FULL_CYCLE_RELOAD_DISABLE                      0x0
#define FULL_CYCLE_RELOAD_ENABLE                       0x1

// Load Mode Select
#define LOAD_NEXT_PWM                                  0x0
#define LOAD_IMMEDIATELY                               0x1
```

```
// Double Switching Enable
#define DOUBLE_SWITCHING_DISABLE                                0x0
#define DOUBLE_SWITCHING_ENABLE                                 0x1

// PWM Prescaler
#define PWM_PRESCALER_FCLOCK                                    0x0
#define PWM_PRESCALER_FCLOCK_2                                  0x1
#define PWM_PRESCALER_FCLOCK_4                                  0x2
#define PWM_PRESCALER_FCLOCK_8                                  0x3
#define PWM_PRESCALER_FCLOCK_16                                 0x4
#define PWM_PRESCALER_FCLOCK_32                                 0x5
#define PWM_PRESCALER_FCLOCK_64                                 0x6
#define PWM_PRESCALER_FCLOCK_128                                0x7

// Load Frequency
#define EVERY_PWM                                               0x0
#define EVERY_2_PWM                                             0x1
#define EVERY_3_PWM                                             0x2
#define EVERY_4_PWM                                             0x3
#define EVERY_5_PWM                                             0x4
#define EVERY_6_PWM                                             0x5
#define EVERY_7_PWM                                             0x6
#define EVERY_8_PWM                                             0x7
#define EVERY_9_PWM                                             0x8
#define EVERY_10_PWM                                            0x9
#define EVERY_11_PWM                                            0xA
#define EVERY_12_PWM                                            0xB
#define EVERY_13_PWM                                            0xC
#define EVERY_14_PWM                                            0xD
#define EVERY_15_PWM                                            0xE
#define EVERY_16_PWM                                            0xF

// PWM Output Polarity
#define PWM_OUTPUT_NOT_INVERTED                                 0x0
#define PWM_OUTPUT_INVERTED                                     0x1

// PWM Fault State
#define LOGIC_0_OUTPUT_POLARITY_CONTROL                         0x0
#define LOGIC_1_OUTPUT_POLARITY_CONTROL                         0x1
#define OUTPUT_IS_TRISTATED                                     0x2

// Reload Error Interrupt Enable
#define REF_CPU_INTERRUPT_REQUESTS_DISABLE                      0x0
#define REF_CPU_INTERRUPT_REQUESTS_ENABLE                       0x1

// Reload Interrupt Enable
#define RF_CPU_INTERRUPT_REQUESTS_DISABLE                       0x0
#define RF_CPU_INTERRUPT_REQUESTS_ENABLE                        0x1

// Capture X Interrupt Enable
#define INTERRUPT_DISABLE_CFX                                   0x0
#define INTERRUPT_ENABLE_CFX                                    0x1

// Compare Interrupt Enables
#define CMPF_NO_INTERRUPT                                       0x0
#define CMPF_INTERRUPT                                          0x1

// Value Registers DMA Enable
#define DMA_WRITE_DISABLE                                       0x0
#define DMA_WRITE_ENABLE                                        0x1
```

```
// FIFO Watermark AND Control
#define FIFO_OR                                                 0x0
#define FIFO_AND                                                0x1

// Capture DMA Enable Source Select
#define READ_DMA_DISABLE                                        0x0
#define FIFO_SET_DMA_READ                                       0x1
#define LOCAL_SYNC_SET_DMA_READ                                 0x2
#define LOCAL_RELOAD_SET_DMA_READ                               0x3

// Output Trigger Enables
#define OUT_TRIG_DISABLE                                        0x00
#define OUT_TRIG_ENABLE                                         0x3F

// Edge Counter X Enable
#define EDGE_COUNTER_DISABLE                                    0x0
#define EDGE_COUNTER_ENABLE                                     0x1

// Input Select X
#define PWMX_INPUT_AS_SORCE                                     0x0
#define OUTPUT_EDGE_COUNTER_AS_SORCE                            0x1

// Edge X
#define DISABLE_EDGE                                            0x0
#define FALLING_EDGES                                           0x1
#define RISING_EDGES                                            0x2
#define ANY_EDGES                                               0x3

// One Shot Mode Aux
#define FREE_RUNNING                                            0x0
#define ONE_SHOT_MODE                                           0x1

// Arm X
#define INPUT_CAPTURE_DISABLE                                   0x0
#define INPUT_CAPTURE_ENABLE                                    0x1

// Edge Compare X
#define compare_value_pwmx                                      0x0

#define SUB_0                                   0x0     // Sub module 0
#define SUB_1                                   0x1     // Sub module 1
#define SUB_2                                   0x2     // Sub module 2
#define SUB_3                                   0x3     // Sub module 3

#define RUN_SUB0                                 0x0100
#define RUN_SUB1                                 0x0200
#define RUN_SUB2                                 0x0400
#define RUN_SUB3                                 0x0800

#define LDOK_SUB0                                0x0001
#define LDOK_SUB1                                0x0002
#define LDOK_SUB2                                0x0004
#define LDOK_SUB3                                0x0008

#define RUN_ON                                                  0xF
#define RUN_OFF                                                 0x0
#define LDOK_ON                                                 0xF
#define LDOK_OFF                                                0x0

/*****************************************************************
* Functions
```

```
*****************************************************************/
void flexpwm0_ch0_init(void);
void flexpwm0_sub0_update_val(uint16_t val2,uint16_t val3,uint16_t
      val4,uint16_t val5);
void flexpwm0_sub0_update_val3(uint16_t val3);
void flexpwm_enable_outputs(void);
void flexpwm_disable_outputs(void);

#endif /* _FLEXPWM_H_ */
```

Nel file "flexpwm.c" sono state sviluppate le funzioni flexpwm0_ch0_init(), flexpwm0_sub0_update_val(), flexpwm0_sub0_update_val3(), flexpwm_enable_outputs() e flexpwm_disable_outputs().

La funzione flexpwm0_ch0_init inizializza il modulo con le impostazioni descritte sopra.

```
void flexpwm0_ch0_init(void)
{
    /*-----------------------------------------------------------------
     * FlexPWM0 Control registers
    -----------------------------------------------------------------*/

      /* OUTEN register: PWMA, PWMB and PWMX are disable */
      FLEXPWM_0.OUTEN.B.PWMA_EN= PWM_DISABLE;
      FLEXPWM_0.OUTEN.B.PWMB_EN= PWM_DISABLE;
      FLEXPWM_0.OUTEN.B.PWMX_EN= PWM_DISABLE;

      /* MASK register: the output of PWMA, PWMB and PWMX are not
      masked*/
      FLEXPWM_0.MASK.B.MASKA= OUTPUT_NORMAL;
      FLEXPWM_0.MASK.B.MASKB= OUTPUT_NORMAL;
      FLEXPWM_0.MASK.B.MASKX= OUTPUT_NORMAL;

      /* SWCOUT register: deadtime are not used, the register are
      reset  */
      FLEXPWM_0.SWCOUT.B.OUT23_3= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT45_3= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT23_2= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT45_2= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT23_1= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT45_1= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT23_0= LOGIC_0_DEADTIME_GENERATOR;
      FLEXPWM_0.SWCOUT.B.OUT45_0= LOGIC_0_DEADTIME_GENERATOR;

       /* DTSRCSEL register: deadtime are not used, the register are
      reset */
      FLEXPWM_0.DTSRCSEL.B.SEL23_3 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL45_3 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL23_2 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL45_2 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL23_1 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL45_1 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL23_0 = GENERATED_PWM_USED_DEADTIME;
      FLEXPWM_0.DTSRCSEL.B.SEL45_0 = GENERATED_PWM_USED_DEADTIME;
```

```
  /* FCTRL register: a logic 1 indicated a fault condition, manual
  fault clearing, safety mode during manual fault clearing, FAULT
  CPU interrupt requests disabled */
  FLEXPWM_0.FCTRL.B.FLVL= LOGIC_1_FAULT;
  FLEXPWM_0.FCTRL.B.FAUTO=MANUAL_FAULT_CLEARING;
  FLEXPWM_0.FCTRL.B.FSAFE=SAFETY_MODE;
  FLEXPWM_0.FCTRL.B.FIE=FAULT_INTERRUPT_DISABLE;

  /* FFILT register: Fault Filter Register are not used, the
  register are reset */
  FLEXPWM_0.FFILT.R= FAULT_GLITCH_STRETCHING_DISABLE;
  FLEXPWM_0.FFILT.B.FILT_CNT= Fault_Filter_Count;
  FLEXPWM_0.FFILT.B.FILT_PER= Fault_Filter_Period;

/*-----------------------------------------------------------------
 * FlexPWM0 sub0 registers
-----------------------------------------------------------------*/
  /* INIT register */
  FLEXPWM_0.SUB[0].INIT.R    = FlexPWM0_INIT_sub0;

  /* CTRL2 register:PWMA and PWMB outputs are independent PWMs,
  IPBus clock  */
  FLEXPWM_0.SUB[0].CTRL2.B.INDEP= INDIPENDENT_PWM;
  FLEXPWM_0.SUB[0].CTRL2.B.INIT_SEL= INITIALIZATION_PWMX;
  FLEXPWM_0.SUB[0].CTRL2.B.FRCEN=INITIALIZATION_FORCE_OUT_DISABLE;
  FLEXPWM_0.SUB[0].CTRL2.B.FORCE_SEL= LOCAL_FORCE_SIGNAL;
  FLEXPWM_0.SUB[0].CTRL2.B.RELOAD_SEL=
                        LOCAL_RELOAD_USED_TO_RELOAD_REGISTERS;
  FLEXPWM_0.SUB[0].CTRL2.B.CLK_SEL= IP_BUS_CLOCK;

  /*CTRL register: Load Frequency every PWM opportunity, Full
  Cycle Reload, Prescaler=0 */
  FLEXPWM_0.SUB[0].CTRL1.B.LDFQ= EVERY_PWM;
  FLEXPWM_0.SUB[0].CTRL1.B.HALF= HALF_CYCLE_RELOAD_DISABLE;
  FLEXPWM_0.SUB[0].CTRL1.B.FULL= FULL_CYCLE_RELOAD_ENABLE;
  FLEXPWM_0.SUB[0].CTRL1.B.PRSC= PWM_PRESCALER_FCLOCK;
  FLEXPWM_0.SUB[0].CTRL1.B.LDMOD= LOAD_NEXT_PWM;
  FLEXPWM_0.SUB[0].CTRL1.B.DBL_EN= DOUBLE_SWITCHING_DISABLE;

  /* VAL0 register */
  FLEXPWM_0.SUB[0].VAL[0].R  = FlexPWM0_VAL_sub0;

  /* VAL1 register */
  FLEXPWM_0.SUB[0].VAL[1].R  = FlexPWM0_VAL1_sub0;

  /* VAL2 register */
  FLEXPWM_0.SUB[0].VAL[2].R  = FlexPWM0_VAL2_sub0;

  /* VAL3 register */
  FLEXPWM_0.SUB[0].VAL[3].R  = FlexPWM0_VAL3_sub0;

  /* VAL4 register */
  FLEXPWM_0.SUB[0].VAL[4].R  = FlexPWM0_VAL4_sub0;

  /* VAL5 register */
  FLEXPWM_0.SUB[0].VAL[5].R  = FlexPWM0_VAL5_sub0;

  /* OCTRL register: PWM output not inverted,Output is forced to
  logic 0 state prior to consideration of output polarity
  control*/
  FLEXPWM_0.SUB[0].OCTRL.B.POLA= PWM_OUTPUT_NOT_INVERTED;
```

```
    FLEXPWM_0.SUB[0].OCTRL.B.POLB= PWM_OUTPUT_NOT_INVERTED;
    FLEXPWM_0.SUB[0].OCTRL.B.POLX= PWM_OUTPUT_NOT_INVERTED;
    FLEXPWM_0.SUB[0].OCTRL.B.PWMAFS=LOGIC_0_OUTPUT_POLARITY_CONTROL;
    FLEXPWM_0.SUB[0].OCTRL.B.PWMBFS=LOGIC_0_OUTPUT_POLARITY_CONTROL;

    FLEXPWM_0.SUB[0].OCTRL.B.PWMXFS=LOGIC_0_OUTPUT_POLARITY_CONTROL;

    /* INTEN register: interrupt are not used, the register are
    reset */
    FLEXPWM_0.SUB[0].INTEN.B.REIE=
                                REF_CPU_INTERRUPT_REQUESTS_DISABLE;
    FLEXPWM_0.SUB[0].INTEN.B.RIE= RF_CPU_INTERRUPT_REQUESTS_DISABLE;
    FLEXPWM_0.SUB[0].INTEN.B.CX1IE= INTERRUPT_DISABLE_CFX;
    FLEXPWM_0.SUB[0].INTEN.B.CX0IE= INTERRUPT_DISABLE_CFX;
    FLEXPWM_0.SUB[0].INTEN.B.CMPIE= CMPF_NO_INTERRUPT;

    /* DMAEN register: DMA write requests disabled,the register are
    reset */
    FLEXPWM_0.SUB[0].DMAEN.B.VALDE= DMA_WRITE_DISABLE;
    FLEXPWM_0.SUB[0].DMAEN.B.FAND= FIFO_OR;
    FLEXPWM_0.SUB[0].DMAEN.B.CAPTDE= READ_DMA_DISABLE;

    /* TCTRL register: Output Trigger disables */
    FLEXPWM_0.SUB[0].TCTRL.B.OUT_TRIG_EN=OUT_TRIG_DISABLE;

    /* DISMAP register: Disable Mapping Register */
    FLEXPWM_0.SUB[0].DISMAP.R  = FlexPWM0_DISMAP_sub0;

    /* DTCNT0 register: deadtime are not used, the register are
    reset */
    FLEXPWM_0.SUB[0].DTCNT0.R  = FlexPWM0_DTCNT0_sub0;

    /* DTCNT1 register: deadtime are not used, the register are
    reset */
    FLEXPWM_0.SUB[0].DTCNT1.R  = FlexPWM0_DTCNT1_sub0;

    /* CAPCTRLX register: Input capture operation is disabled, the
    register are reset */
    FLEXPWM_0.SUB[0].CAPTCTRLX.B.EDGCNTXEN= EDGE_COUNTER_DISABLE;
    FLEXPWM_0.SUB[0].CAPTCTRLX.B.INPSELX= PWMX_INPUT_AS_SORCE;
    FLEXPWM_0.SUB[0].CAPTCTRLX.B.EDGX1= DISABLE_EDGE;
    FLEXPWM_0.SUB[0].CAPTCTRLX.B.EDGX0= DISABLE_EDGE;
    FLEXPWM_0.SUB[0].CAPTCTRLX.B.ONESHOTX= FREE_RUNNING;
    FLEXPWM_0.SUB[0].CAPTCTRLX.B.ARMX= INPUT_CAPTURE_DISABLE;

    /* CAPTCOMPX register: Input capture operation is disabled, the
    register are reset */
    FLEXPWM_0.SUB[0].CAPTCOMPX.B.EDGCMPX= compare_value_pwmx;

    /* MCTRL register: Load prescaler, modulus and PWM values,PWM
    generator enabled */
    FLEXPWM_0.MCTRL.B.LDOK= LDOK_ON;
    FLEXPWM_0.MCTRL.B.RUN= RUN_ON;


}
```

Le funzioni flexpwm0_sub0_update_val e flexpwm0_sub0_update_val3 servono per variare i valori degli istanti dei fronti. La prima permette di cambiare VAL2, VAL3, VAL4 e VAL5 mentre la seconda permette di cambiare solamente VAL3.

```
void  flexpwm0_sub0_update_val(uint16_t  val2,uint16_t  val3,uint16_t
val4,uint16_t val5)
{

      FLEXPWM_0.MCTRL.B.LDOK= LDOK_OFF;

      FLEXPWM_0.SUB[0].VAL[2].R  = val2;
      FLEXPWM_0.SUB[0].VAL[3].R  = val3;
      FLEXPWM_0.SUB[0].VAL[4].R  = val4;
      FLEXPWM_0.SUB[0].VAL[5].R  = val5;

      FLEXPWM_0.MCTRL.B.LDOK= LDOK_ON;

}

void flexpwm0_sub0_update_val3(uint16_t val3)
{

      FLEXPWM_0.MCTRL.B.LDOK= LDOK_OFF;
      FLEXPWM_0.SUB[0].VAL[3].R = val3;
      FLEXPWM_0.MCTRL.B.LDOK= LDOK_ON;

}
```

Le funzioni flexpwm_enable_outputs e flexpwm_disable_outputs servono per abilitare la generazione del segnale PWM e per disabilitarla.

```
void flexpwm_enable_outputs()
{
      FLEXPWM_0.OUTEN.B.PWMA_EN= PWM_ENABLE;
      FLEXPWM_0.OUTEN.B.PWMB_EN= PWM_ENABLE;
      FLEXPWM_0.OUTEN.B.PWMX_EN= PWM_ENABLE;

}

void flexpwm_disable_outputs()
{
      FLEXPWM_0.OUTEN.B.PWMA_EN= PWM_DISABLE;
      FLEXPWM_0.OUTEN.B.PWMB_EN= PWM_DISABLE;
      FLEXPWM_0.OUTEN.B.PWMX_EN= PWM_DISABLE;

}
```

A.4 Istruzioni in linguaggio C per la lettura di un potenziometro

Il seguente codice permette di leggere un ingresso, che nel nostro caso era un potenziometro. È stata creata una funzione chiamata initADC dove sono stati inizializzati i registri. Nel registro MCR si è impostato che la conversione fosse continua, che il nuovo valore della conversione venga sovrascritto al valore di prima, il clock uguale al system clock e che l'allineamento venga fatto a destra. È stata abilitata la normal conversion sul canale 0 e disabilitata la injected conversion.

```
void initADC(void)
{
      ADC_0.MCR.B.PWDN = 0;       // Disable the power down mode
      ADC_0.MCR.B.MODE   = 1;     // Initialize for Scan Mode
      ADC_0.MCR.B.OWREN  = 1;     // Enables converted data to be
                                    overwritten by a new conversion
      ADC_0.MCR.B.ADCLKSEL  = 1; // ADC clock is equal the sys clock
      ADC_0.MCR.B.WLSIDE = 0;     // Write right aligned

      /* Mask ADC0 interrupts */
      ADC_0.CIMR[0].R = 0x00000000;

      /* Normal Conversion Masked Channels */
      ADC_0.NCMR[0].R = 0x00000001;   // sampling enabled for AN0

      /* Injected Conversion Masked Channels */
      ADC_0.JCMR[0].R = 0x00000000;

      /* Conversion times configuration */
      ADC_0.CTR0.R  = 0x00008606;


}
```

La conversione ha inizio quando nel main si imposta il campo NSTART del registro MCR a 1.

```
  ADC_0.MCR.B.NSTART = 1;          // start conversion ADC
```

A.5 Istruzioni in linguaggio C per inizializzare le periferiche e il PIT

In questo progetto si è implementata una funzione, chiamata initGPIO, dove sono state configurate tutti i pin di ingresso e di uscita. Sono stati impostati i pin delle porte A[0], A[1] e C[11], che corrispondono al canale 0, 1 e 4 dell'eTimer 0, come input dell'encoder. Inoltre per la porta A[43] si è dovuto impostare il registro PSMI[7] a 1 per associare il canale 4 dell'eTimer 0 alla porta C[11]. Si è impostato come uscite le porte A[10] e A[11], per il canale 0 del flexPWM 0, e G[2] che è un led della scheda. La porta B[7] è stata configurata come ingresso analogico del ADC 0 canale 0, mentre le porte A[18] e A[19] sono state configurate una come ingresso e l'altra come uscita per la comunicazione con una porta seriale.

```
void InitGPIO(void)
{

SIU.PCR[0].R = 0x0500;//Pin port A[0]in ALT1 mode: Timer 0, channel 0
SIU.PCR[1].R = 0x0500;//Pin port A[1] in ALT1 mode: Timer 0, channel 1
SIU.PSMI[7].B.PADSEL=1;//Pad Selection 7 to Timer 0 channel 4 in pin43
SIU.PCR[43].R = 0x0500;//Pin port C[11]in ALT1 mode:Timer 0, channel 4
SIU.PCR[11].R = 0x0A00;// Pin port A[11] in FlexPWM A[0] output
SIU.PCR[10].R = 0x0A00;// Pin port A[10] in FlexPWM B[0] output
SIU.PCR[23].R = 0x2000;//Pin port B[7] in Analog input mode: ADC 0 ch0
SIU.PCR[98].R = 0x0200;     //Pin port G[2] is selected in GPIO output
SIU.PCR[18].R = 0x0604;     // Configure pad B[2] for LIN_0 TX
SIU.PCR[19].R = 0x0100;     // Configure pad B[3] for LIN_0 RX

}
```

Nella seguente funzione si inizializza il modulo PIT per fare due interrupt: uno ogni 1ms, per il controllo dell'uscita, e l'altro ogni 200ms utilizzato per comunicare dei dati attraverso la seriale. Si è caricato 119999 nel PIT0 e 23999999 nel PIT1, in quanto il clock del modulo è il system clock che è impostato a 120MHz.

```
void Pit_enable(void)
{
      //Enable PIT
      PIT.PITMCR.R=0x00;

      //Enable PIT0 every 1 ms
      PIT.LDVAL0.R=0x0001D4BF; //Load value 119999
      PIT.TCTRL0.B.TIE=0x1;
      PIT.TCTRL0.B.TEN=0x1;

      //Enable PIT1 every 200ms
      PIT.LDVAL1.R=0x016E35FF; //Load value 23999999
      PIT.TCTRL1.B.TIE=0x1;
      PIT.TCTRL1.B.TEN=0x1;


}
```

A. Istruzioni in linguaggio C per il controllo di posizione di un motore

Nel seguente codice si è implementato il controllo di posizione descritto nel capitolo 4. Nel main inizialmente si sono inizializzati tutti i moduli (eTimer, FlexPWM, ADC, GPIO, LinFlex) e il clock. Inoltre si abilita l'uscita del segnale PWM impostato a 0 (duty cycle a 50% e quindi la tensione al motore è uguale a 0). Si è fatto in modo che premendo il PAD3 si abilitano i PIT anche se i controlli sono disabilitati. Premendo il PAD2 inizia il controllo di velocità: questo controllo è stato inserito per portare il motore nello "zero" encoder. L'encoder quando arriva sullo zero crea un interrupt che va a disabilitare il controllo di velocità. Successivamente premendo il PAD3 si abilita il controllo di posizione.

```
int main(void)
{

  /* All Master Can go through AIPS and all peripheral have no
protection */
  AIPS.MPROT0_7.R       = 0x77777777;
  AIPS.MPROT8_15.R      = 0x77000000;
  AIPS.PACR0_7.R        = 0x0;
  AIPS.PACR8_15.R       = 0x0;
  AIPS.PACR16_23.R      = 0x0;
  AIPS.OPACR0_7.R       = 0x0;
  AIPS.OPACR16_23.R     = 0x0;
  AIPS.OPACR24_31.R     = 0x0;
  AIPS.OPACR32_39.R     = 0x0;
  AIPS.OPACR40_47.R     = 0x0;
  AIPS.OPACR48_55.R     = 0x0;
  AIPS.OPACR56_63.R     = 0x0;
  AIPS.OPACR64_71.R     = 0x0;
  AIPS.OPACR80_87.R     = 0x0;
  AIPS.OPACR88_95.R     = 0x0;

  //Enable pad
  INT_PAD2_ENABLE;
  INT_PAD3_ENABLE;

  //Initialization device
  InitHW();
  InitGPIO();
  init_LinFLEX_0_UART();
  etimer0_init();
  flexpwm0_ch0_init();
  initADC();

  //turn off test interrupt LED
  SIUL.GPDO[98].R = 0x1;

  printf("Device ready!\n\n");

  //Enable PWM output
  flexpwm_enable_outputs();

  while (INT_PAD3_INACTIVE) {};
```

```
  //Disable speed and position control
  cntr_pos_enable = 0;
  cntr_speed_enable = 0;

  ADC_0.MCR.B.NSTART = 1;       // start conversion ADC
  Pit_enable();                 //Enable PIT interrupt

  //Start PIT interrupt
  INTC_InstallINTCInterruptHandler(CONTROL_OUT, 59, 3);
  INTC_InstallINTCInterruptHandler(PRINT_DATA, 60, 1);

  /* lower current INTC priority to start INTC interrupts */
  INTC.CPR_PRC0.R = 0;

  while (INT_PAD2_INACTIVE) {};

  //Enable speed control
  cntr_speed_enable = 1;
  printf("\nControl speed Enable!\n\n");

  /*STS register: clear flag IEHF*/
  ETIMER_0.CHANNEL[1].STS.B.IEHF = 1;

  //Start eTimer interrupt
  INTC_InstallINTCInterruptHandler(DISABLE_SPEED_CONTROL, 158, 2);

  while (INT_PAD3_INACTIVE) {};

  //Enable position control
  cntr_pos_enable = 1;
  printf("\nControl position Enable!\n\n");


  while (1) {

            };


}
```

La funzione INTC_InstallINTCInterruptHandler(handlerFn, vectorNum, psrPriority) permette di eseguire la funzione handlerFn quando si è verificato un interrupt generato dalla sorgente che è associata al numero vectorNum. Il numero psrPriority assegna la priorità dell'interrupt: il microcontrollore infatti prevede 16 priorità.

La funzione DISABLE_SPEED_CONTROL viene eseguita quando si verifica l'interrupt dovuto all'encoder. Quando il canale 1 dell'eTimer trova un fronte di salita, e quindi è sullo "zero" encoder, si genera l'interrupt. Questa funzione disabilita il controllo di velocità e l'interrupt dovuto all'encoder.

```
static void DISABLE_SPEED_CONTROL (void)
{
      //Disable speed control
```

```
	cntr_speed_enable=0;

	//Clear flag interrupt
	ETIMER_0.CHANNEL[1].STS.B.IEHF=1;

	//Disable interrupt eTimer
	ETIMER_0.CHANNEL[1].INTDMA.B.IEHFIE=0;
}
```

La funzione CONTROL_OUT viene eseguita ogni 1ms e serve per aggiornare l'uscita PWM. Inizialmente vengono letti e calcolati i nuovi dati come posizione(cont), errore di posizione(q) e velocità(q_dot) convertendoli quest'ultimi due in rad e rad/s. In seguito sono implementati i controlli di velocità e di posizione. Nel controllo di velocità si limita la velocità a 5 rad/s fino a quando l'interrupt dovuto all'encoder disattiva questo controllo. Nel controllo di posizione il riferimento viene fatto variare con il potenziometro in ingresso dell'ADC. Dopo aver caricato il valore attuale nel valore dell'integratore ed aver fatto una semplice anti-wind up dove limito il valore dell'integratore, calcolo l'uscita per il segnale PWM con il controllo PID con le costanti K_P, K_i e K_D calcolate sopra. L'uscita del segnale PWM è stata limitata ai valori 307 e -307 che corrispondono a ±3A, questo perché il motore può ricevere al massimo quel valore di corrente. A questo punto l'uscita viene modificata con i dati calcolati, e vengono aggiornati i dati in memoria.

```
static void CONTROL_OUT(void)
{
	SIUL.GPDO[98].R ^= 0x1; //toggle  test interrupt LED

	// Data Read
	cont     = (int16_t)value_eTimer();
	adc_cont = (int16_t)ADC_0.CDR[0].B.CDATA;

	//Calcolate data
	adc_sum  = adc_sum + adc_cont;
	q        = 3.14*((float)(cont-(adc_ref-2048)))/1000;
	q_dot    = 166*(q +3*q1-3*q2-q3);

	//Control Update
	cntr_output = 0;

	if (cntr_speed_enable){

		//Calculate output for the speed control
		cntr_output = (int32_t)((q_dot-5)*50);

	}

	if (cntr_pos_enable){


		qi = q*0.001+qi;		//Store value for integral
```

```
            if (qi>400000) qi =400000;    //Antiwind up

            //Value PID
            Kp =10.3867*2048/10;
            Ki =49.1691*2048/10;
            Kd =0.1463*2048/10;

            //Calcolate output PID
            cntr_output = (int32_t)(Kp*q+Kd*q_dot+Ki*qi);

      }

      //Limit output to motor to 3A
      if(cntr_output>=307)
      {
            cntr_output=307;
      }else if (cntr_output<=-307)
            cntr_output=-307;

      //Change output
      flexpwm0_sub0_update_val3((int16_t)cntr_output);

      // Data to  serial
      pos     = (int32_t)(q*1000);
      speed   = (int32_t)(q_dot*1000);

      // Update data
      q3      = q2;
      q3_dot  = q2_dot;
      q2      = q1;
      q2_dot  = q1_dot;
      q1      = q;
      q1_dot  = q_dot;

      //Clear flag interrupt
      PIT.TFLG0.B.TIF = 1;
}
```

Inoltre è stata creata una funzione PRINT_DATA per comunicare con la porta seriale e visualizzare i dati nel Terminal di CodeWarrior: i dati vengono aggiornati ogni 200ms ed è particolarmente utile in fase di sviluppo. Per il riferimento non si è utilizzato direttamente il valore del potenziometro ma si è fatto una media: in questo modo il valore risulta più stabile.

```
static void PRINT_DATA(void)
{
      //Print to serial
      printf("\rerr:%d\t",pos);
      printf("\tvel:%d ",speed);

      //Calculate average of adc value
      adc_ref = adc_sum/200;
      adc_sum = 0;

      //Clear flag interrupt
      PIT.TFLG1.B.TIF = 1;

}
```