Università degli Studi di Padova

Facoltà di Ingegneria

Dipartimento di Tecnica e Gestione dei Sistemi Industriali

Corso di laurea triennale in Ingegneria

Meccanica e Meccatronica

Curriculum Meccatronico

Tesi di Laurea

Gestione e ottimizzazione del software per il controllo di un veicolo autonomo

Laureando:

Cattapan Nicola

1010657 – IMM

Relatore:

Ch.mo Prof.

Roberto Oboe

Anno Accademico: 2013 – 2014

Sommario

La Freescale Cup oltre ad essere una gara tra studenti di tutto il mondo è un'ottima opportunità per entrare in un'ottica lavorativa ancora sconosciuta a noi studenti. L'Università di Padova, sotto la guida del professor Roberto Oboe, ha deciso di proporre a noi studenti del secondo e terzo anno di Ingegneria Meccatronica questa competizione come occasione per mettere alla prova le nostre conoscenze confrontandoci con concorrenti italiani e stranieri. Questa esperienza ci ha insegnato a collaborare lavorando in team, imparare l'uno dall'altro e crescere professionalmente attraverso lo scambio reciproco di idee ed esperienze anche con i nostri colleghi ed "avversari". Inoltre abbiamo approfondito le nostre conoscenze sul funzionamento dei microcontrollori e sulla microelettronica risolvendo problemi pratici difficilmente affrontabili in aule universitarie.
Per concorrere al meglio in questa competizione sono state necessarie competenze di: programmazione in linguaggio C di sistemi embedded, controlli automatici, microelettronica, fondamenti di macchine ed azionamenti elettrici, misure per l'automazione, teoria dei circuiti digitali, dinamica dei veicoli ed elaborazione e utilizzo di dati provenienti dal blocco sensori.
È stata un'esperienza formativa lunga e impegnativa sia dal punto di vista dello studio che da quello del lavoro in team, ma che sicuramente consiglio agli studenti dell'ambito meccatronico che vogliano mettersi alla prova con i veri problemi "del mestiere" poiché aiuterà un giorno ad entrare nel mondo del lavoro più competenti e preparati.

Indice

Capitolo 1

Introduzione

1.1 FREESCALE CUP

La Freescale Cup è una competizione mondiale per studenti di Ingegneria che si svolge in ventiquattro paesi di tutto il mondo (Malesia, Cina, Corea, India, Giappone, Taiwan, Brasile, Messico, Stati Uniti e quattordici stati in tutta Europa) per un totale di oltre 32.000 studenti partecipanti ogni anno. Questa gara organizzata dall'azienda Freescale Semiconductor™ consiste nel costruire, assemblare, programmare e testare una smart car (mini vettura) in scala 1/18 capace di seguire qualsiasi tracciato in modo totalmente automatico e autonomo senza alcun intervento di un utente esterno.

Figura 1.1 – Logo della competizione

La pista è caratterizzata da pannelli bianchi in materiale ABS larghi 50 cm con nel mezzo una linea guida nera larga 2.5 cm da seguire per tutta la durata del giro. Il circuito consiste in un percorso chiuso composto da una sequenza non prestabilita di rettilinei, curve di raggio minimo 50 cm, *chicane*, incroci a 90°, dossi di massimi 15° di pendenza, bande sonore e tunnel in modo da testare l'affidabilità delle smart car create.
Vince la competizione chi riesce a far compiere un giro completo del circuito al proprio elaborato partendo dietro un particolare segnale di stop e fermandosi entro i 3 metri successivi questo segnale, dopo aver percorso il tracciato nel minore tempo possibile e senza uscire dalla pista.

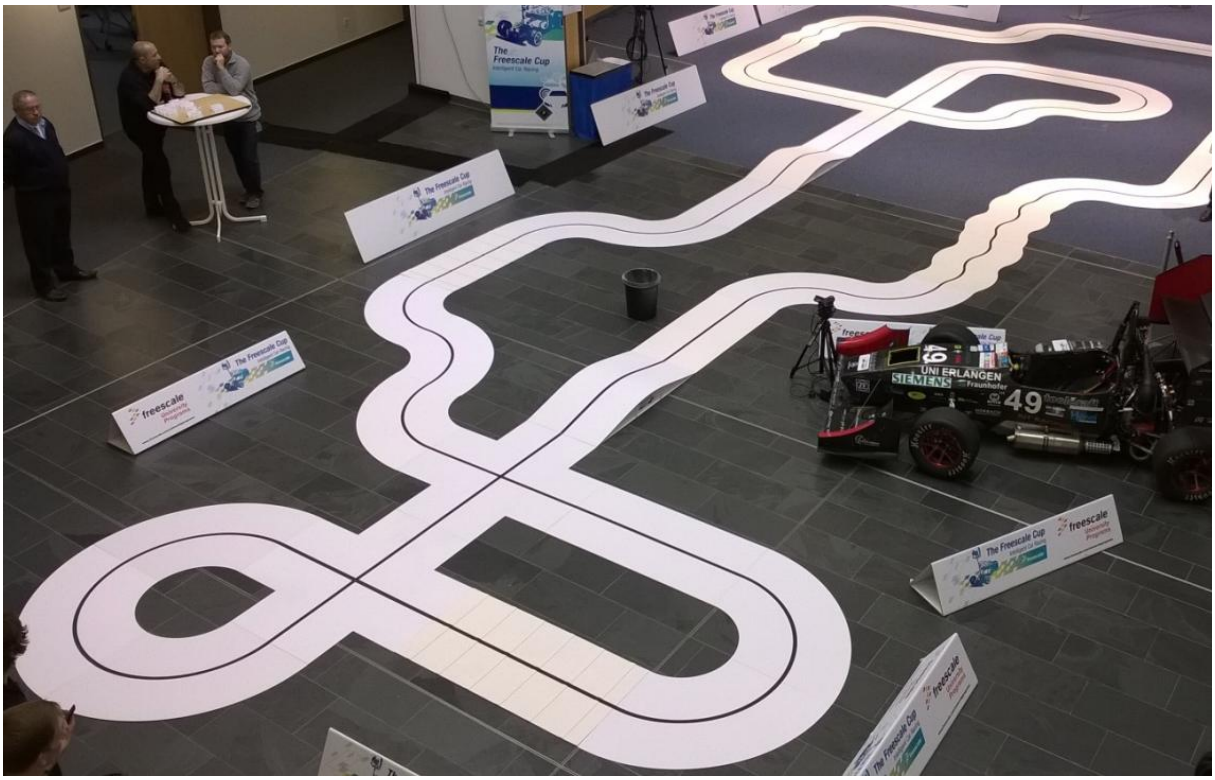


Figura 1.2 – Circuito ufficiale delle finali europee 2014

Per creare queste smart car si deve lavorare in team, formato da minimo due e massimo quattro componenti.
L'azienda per proporre questa esperienza formativa fornisce ad ogni team partecipante un kit di materiale base per costruire le smart car, contenente:

- uno chassis
- ruote e pneumatici
- sospensioni
- due motori in corrente continua
- una scheda di potenza con due ponti ad H
- una trasmissione
- un servomotore
- una batteria da 7.2V, ≤3000mAh
- un microcontrollore
- una telecamera lineare

I vari team possono modificare e integrare questo kit aggiungendo ulteriori sensori ausiliari per migliorare velocità, stabilità ed affidabilità del veicolo.

1.2 STRUTTURA DELL'ELABORATO

In questo elaborato si vuole discutere velocemente l'hardware del veicolo per poi capire dettagliatamente tutti i concetti relativi all'implementazione software. È certamente necessaria un'introduzione relativa all'hardware perché senza di essa risulterebbe impossibile comprendere in profondità la logica di comando e di controllo attuata nel programma creato. Si vuole fornire con il presente documento uno strumento di comprensione per gli studenti che parteciperanno alla competizione Freescale Cup negli anni futuri, con la speranza che risulti loro utile.
Il sistema in esame è il classico sistema meccatronico dove sono richieste competenze nell'ambito elettromeccanico, elettronico ed in particolare una notevole dimestichezza con i linguaggi di programmazione sia di alto che basso livello.
Di seguito sarà trattata la suddivisione dell'elaborato per capitoli, riassumendone il contenuto in modo da dare al lettore la possibilità di una maggiore visione dell'insieme. Ogni capitolo sarà accompagnato da immagini per facilitarne la comprensione.

Capitolo 2: Hardware

In questo capitolo sarà descritto il kit fornito dall'azienda promotrice Freescale Semiconductor™: si parlerà inizialmente dell'unità microcontrollore utilizzata e della sua architettura, per poi soffermarsi sui sensori e gli attuatori utilizzati.
Dei due sensori impiegati partiremo con la trattazione di quello che è l'unica interfaccia con l'ambiente esterno e che mantiene sempre monitorata la posizione della linea guida del tracciato che si vuole percorrere: la Line Scan Camera TSL1401CL prodotta dalla TAOS®. Ne sarà discusso il funzionamento e i principali disturbi esterni riscontrati durante il suo utilizzo nel segnale d'uscita. Poi si passerà a parlare dell'encoder: questo sensore è stato da noi realizzato e per questo ne verranno illustrati i componenti, la circuiteria implementata e i segnali uscenti da tale periferica.
Successivamente ci si soffermerà sugli attuatori utilizzati. Il primo che si incontrerà è lo stadio di potenza nella Motor Drive Board con la quale si comandano separatamente i due motori in corrente continua del tipo Standard Motor: "rn 260 c" winding 18130 mediante due ponti ad H MC33931. Il secondo è il Servo Motor Futaba S3010 del quale si spiegheranno le tecniche di comando per l'attuazione di un preciso angolo di sterzata.

Capitolo 3: Software

Questo è il capitolo principale dove saranno discusse la gestione e l'ottimizzazione del software, spiegando i problemi incontrati e le soluzioni da noi adottate. Il programma è stato diviso in più *files* sorgenti, quindi prima si spiegheranno i metodi ausiliari allo svolgimento del programma, poi quelli relativi alla telecamera, all'interfacciamento e all'inizializzazione dei canali del microprocessore, allo sterzo, ai motori ed infine il programma principale Main.c.
I metodi ausiliari sono due: uno relativo ai valori delle variabili ed uno di dichiarazione dei ritardi, entrambi utilizzati in tutto il programma. In Telecamera.c si trovano le funzioni di inizializzazione della camera, di ricerca della linea nera e del segnale di stop mediante il metodo derivativo. In Pinout.c si inizializzano porte, ADC e moduli e canali eMIOS del microcontrollore. In Sterzo.c invece si incontrano i metodi di impostazione dell'angolo di sterzo mediante PWM e di implementazione del controllore PID. In Motori.c si comandano i ponti ad H, si utilizza il controllore PID e si elabora un differenziale elettronico. Infine in Main.c si trovano le funzioni relative alla generazione del clock di sistema, all'inizializzazione degli *interrupt* dei PID di sterzo e motori e alla lettura dagli encoder; inoltre si sfrutta un ciclo while infinito per continuare ad aggiornare le variabili di sistema.

Capitolo 2

Hardware

Per poter comprendere pienamente la gestione e l'ottimizzazione del software utilizzato per comandare questo veicolo autonomo è necessaria una veloce, ma dettagliata, trattazione dell'hardware della smart car, mettendo in evidenza i limiti dei sensori e gli eventuali disturbi esterni che potrebbero compromettere la stabilità del sistema. Con questo obiettivo scopriremo l'unità microcontrollore, i sensori montati (telecamera ed encoder) e gli attuatori necessari (servo motor e blocco motori).

2.1 MICRO CONTROLLER UNIT

Se il veicolo autonomo può essere descritto come un essere umano, allora l'unità microcontrollore ne è il cervello. Il kit comprende il modello TRK MCB5604B prodotto da Freescale Semiconductor™. Esso rappresenta una nuova generazione di microcontrollori a 32 bit basati su Power Architecture® ed appartiene ad una famiglia più ampia di prodotti ideati per applicazioni *automotive*, mirati ad affrontare e gestire il numero sempre in crescita di applicazioni elettroniche a bordo del veicolo. Questo microcontrollore lavora a 64 MHz ed offre alte prestazioni di processing ottimizzate per il consumo a bassa energia.

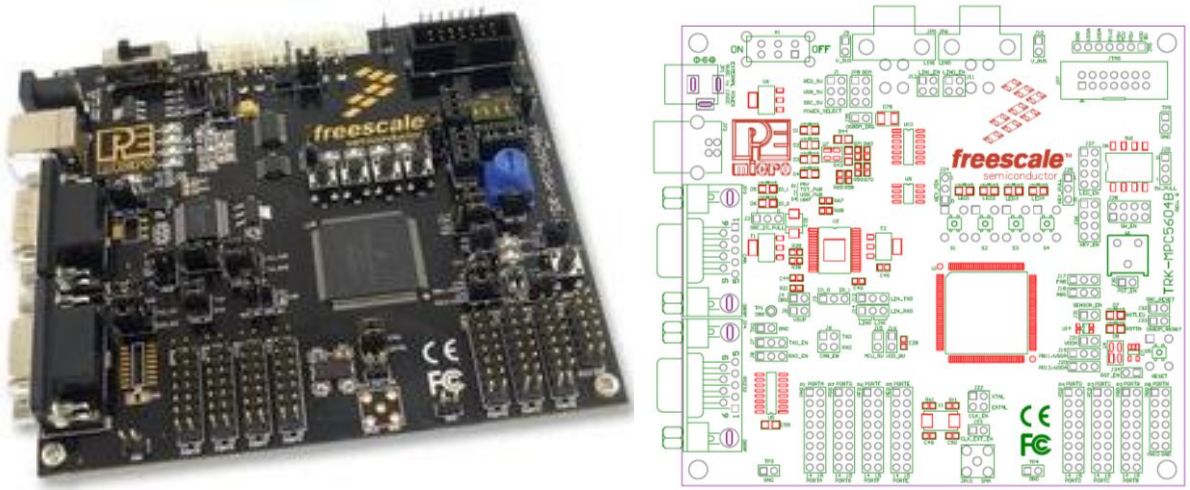


Figura 2.1 – Microcontrollore TRK MCB5604B

Dal data sheet relativo, possiamo ricavarne le seguenti specifiche:

CORE:
PowerPC e200z0 core running 48-64MHz
VLE ISA instruction set for superior code density
Vectored interrupt controller
Memory Protection Unit with 8 regions, 32 byte granularity

MEMORY :
512 Kbyte embedded program Flash, 64 KByte data flash
64 Kbyte embedded data Flash (for EE Emulation)
Up to 64MHz non-sequential access with 2WS
ECC-enabled array with error detect/correct
48 Kbyte SRAM (single cycle access, ECC-enabled)

COMMUNICA TIONS :
3x enhanced FlexCAN
64 Message Buffers each, full CAN 2.0 spec
4x LINFlex
3x DSPI, 8-16 bits wide & chip selects
1xI2C

ANALOG :
5V ADC 10-bit resolution

TIMED I/O :
16-bit eMIOS module

OTHER:
CTU (Cross Triggering Unit) to sync ADC with PWM Channels
I/O: 5V I/O, high flexibility with selecting GPIO functionality
Packages: 100LQFP, 144LQFP, 208MAPBGA (Development only)
Boot Assist Module for production and bench programming

I principali clock di sistema provengono da tre sorgenti:
- Fast External Oscillator 4-16 MHz (FXOSC)
- Fast Internal RC Oscillator 16 MHz (FIRC)
- Frequency Modulated Phase Locked Loop (FMPLL)

Il microcontrollore è dotato inoltre di ulteriori due oscillatori a bassa potenza:
- Slow Internal RC Oscillator 128 kHz (SIRC)
- Slow External Crystal Oscillator 32 kHz (SXOSC)

2.1.1 Architettura dei moduli eMIOS

Il microcontrollore dispone di moduli eMIOS (enhanced Modular IO Subsystem) che offrono molte funzionalità specifiche per generare o misurare eventi temporali. Ogni modulo offre una combinazione di uscite ed ingressi con funzioni di comparazione e funzionalità di tipo PWM (Pulse Width Modulation): una particolare tecnica che permette di generare segnali ad onda quadra caratterizzati da un particolare duty cycle.

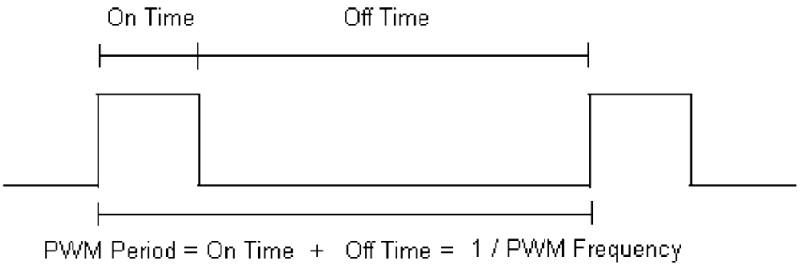


Figura 2.2 – Schematizzazione di un periodo di PWM

$$\text{Duty cycle: } \delta = \frac{T_{on}}{T_{on} + T_{off}}$$

Il T_{on} rappresenta la durata dell'intervallo temporale in cui il segnale è alto, mentre durante il T_{off} il segnale risulta basso.

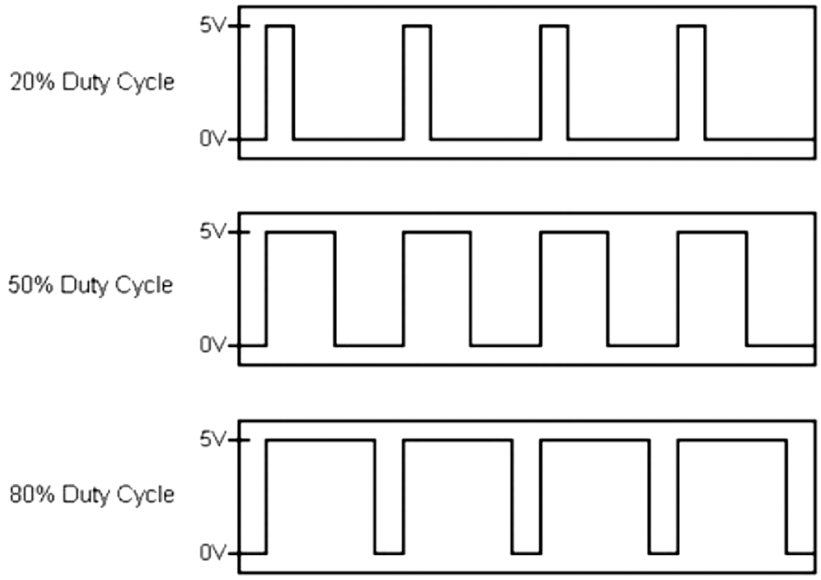
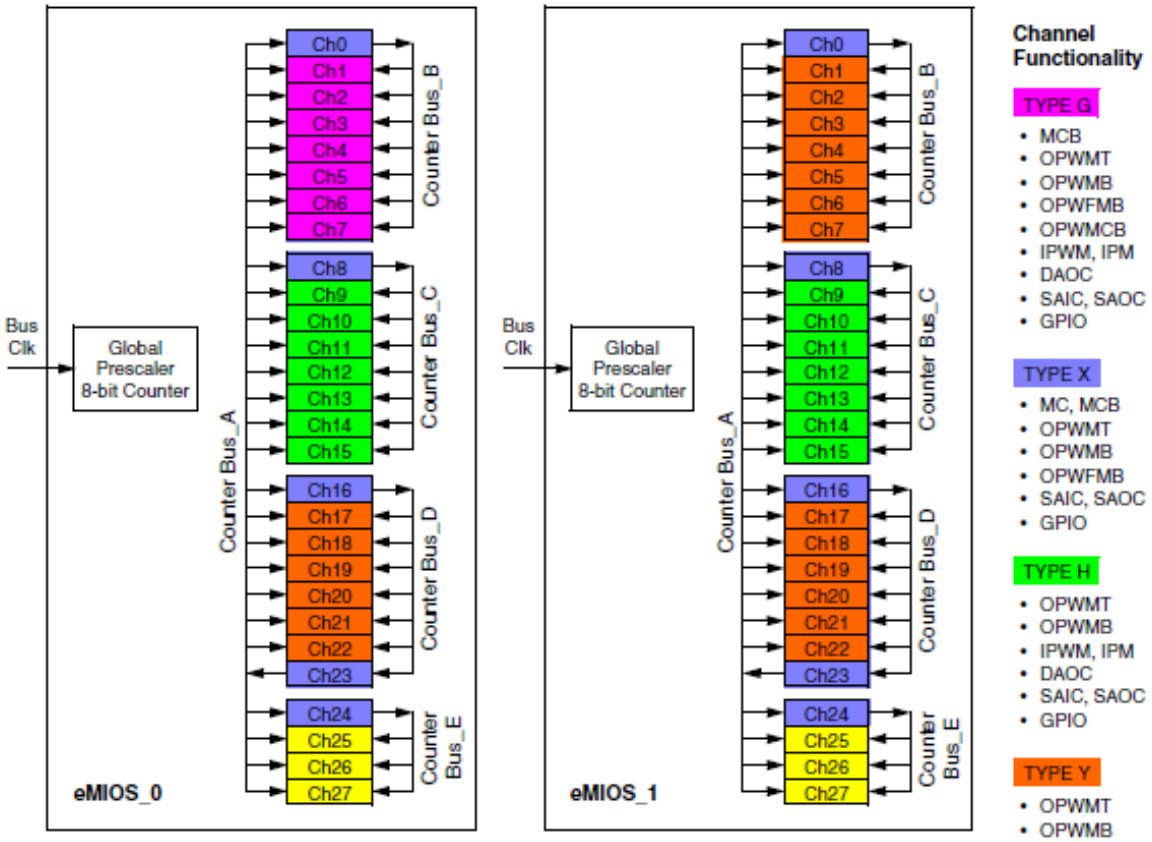


Figura 2.3 – Esempi di diversi duty cycle

Il controllore ha implementato diversi tipi di canali, inoltre le funzionalità variano a seconda del modulo eMIOS che si sta considerando. Ogni canale al suo interno dispone di un proprio *internal counter* che garantisce una risoluzione di 16 bit. Per permettere la sincronizzazione tra canali diversi si dispone di un certo numero di counter buses che possono essere utilizzati per avere un riferimento temporale comune. I counter buses possono essere utilizzati assieme ai contatori individuali dei canali per implementare funzionalità avanzate. I moduli eMIOS ricevono in ingresso i SYSCLK a 64 MHz, assieme alle altre periferiche appartenenti al Peripheral Set 3 (eMIOS, ADC, CTUL).

Dal data sheet, possiamo ricavare le seguenti specifiche dei moduli eMIOS:
- 2 blocchi eMIOS da 28 canali ciascuno
- 50 canali dispongono di funzionalità di tipo Output PWM Triggered (OPWMT)
- 6 canali con funzioni esclusivamente di tipo IC/OC
- un prescaler globale
- registri a 16 bit
- 10 counter buses a 16 bit
- i contatori B,C,D,E possono essere comandati solo dai Unified Channel 0, 8, 16 e 24 rispettivamente
- il contatore A può essere comandato solo dal Canale 23.

I Canali eMIOS possono essere configurati via software per operare in uno dei seguenti modi come illustrato in figura 2.4:



Key

DAOC	Dual Action Output Compare
GPIO	General Purpose Input Output
IPM	Input Period Measurement
IPWM	Input Pulse Width Measurement
MC	Modulus Counter
MCB	Buffered Modulus Counter
OPWMB	Buffered Output Pulse Width Modulation
OPWMT	Buffered Output Pulse Width Modulation with Trigger
OPWFMB	Buffered Output Pulse Width and Frequency Modulation
OPWMCB	Center Aligned Output PWM Buffered with Dead-Time
SAIC	Single Action Input Capture
SAOC	Single Action Output Compare

Figura 2.4 – Diverse modalità di funzionamento dei canali eMIOS

Le modalità di funzionamento dei canali possono essere impostate tramite i *mode selection bits* MODE[0:6] negli *eMIOS UC Control Register* come evidenziato in figura 2.5:

MODE[0:6]	Mode of operation
000_0000	General Purpose Input/Output (input)
000_0001	General Purpose Input/Output (output)
000_0010	Single Action Input Capture
000_0011	Single Action Output Compare
000_0100	Input Pulse width Measurement
000_0101	Input Period Measurement
000_011b	Double Action Output compare
000_1000 000_1111	Reserved
001_0bbb	Modulus Counter
001_1000 010_0101	Reserved
010_0110	Output Pulse Width Modulation with Trigger
010_0111 100_1111	Reserved
101_000b	Modulus Counter Buffered (Up counter)
101_0010 101_0011	Reserved
101_010b	Modulus Counter Buffered (Up/Down counter)
101_0110 101_1001	Reserved
101_10b0	Output Pulse Width and Frequency Modulation Buffered
101_11bb	Center Aligned Output Pulse Width Modulation Buffered
110_00b0	Output Pulse Width Modulation Buffered
110_0100 111_1111	Reserved

Figura 2.5 – Tabella delle modalità di funzionamento dei canali

2.2 SENSORI

I sensori sono elementi fondamentali di un qualsiasi sistema di acquisizione dati o di misurazione: essi forniscono in uscita un segnale elettrico legato alla grandezza fisica in ingresso. Nel nostro caso sono la telecamera e i due encoder.

2.2.1 LINE SCAN CAMERA

Il sensore principale della smart car è la Line Scan Camera: è quello che la interfaccia con l'ambiente esterno mantenendo sempre monitorata la posizione della linea guida del tracciato precedente ad essa. Quella fornitaci dalla Freescale Semiconductor™ è la TSL1401CL prodotta dalla TAOS®: essa è costituita da una matrice 1x128 di sensori CMOS, abbinati a degli amplificatori operazionali di carica, a delle funzionalità di *data hold* che permettono di attivare e disattivare simultaneamente l'integrazione per tutti i pixel e da un fuoco regolabile con una lente di 7.9 mm. I 128 fotodiodi sono disposti in linea con una superficie individuale di 3524.3 μm², distanziati tra loro di 8 μm.

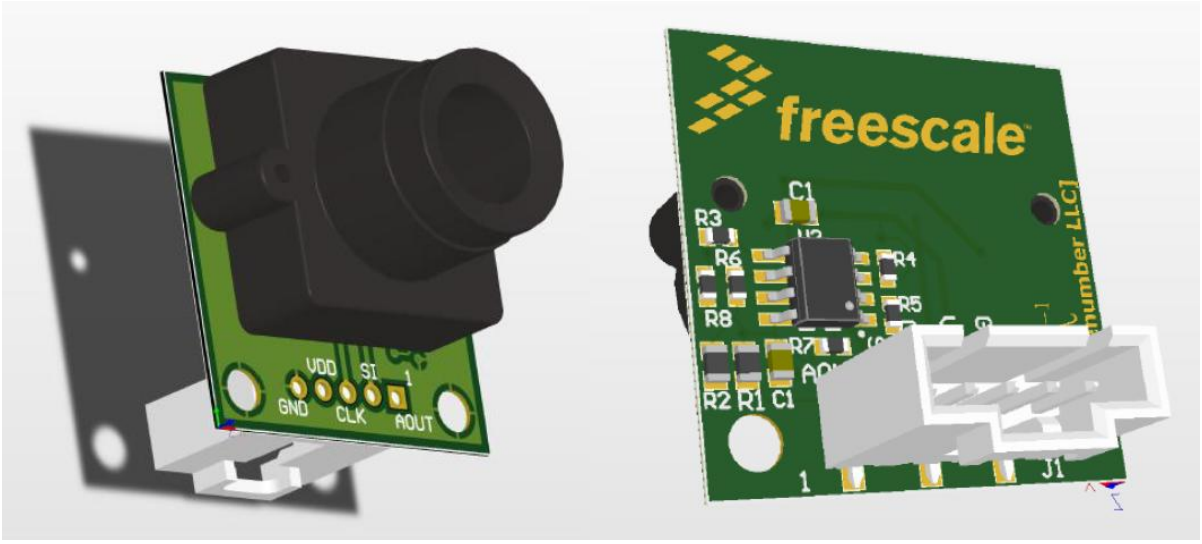


Figura 2.6 – Sensore TSL1401CL

Per capirne i problemi legati ai possibili disturbi che si possono incontrare durante la fase di test è necessario analizzare lo schema funzionale del sensore:

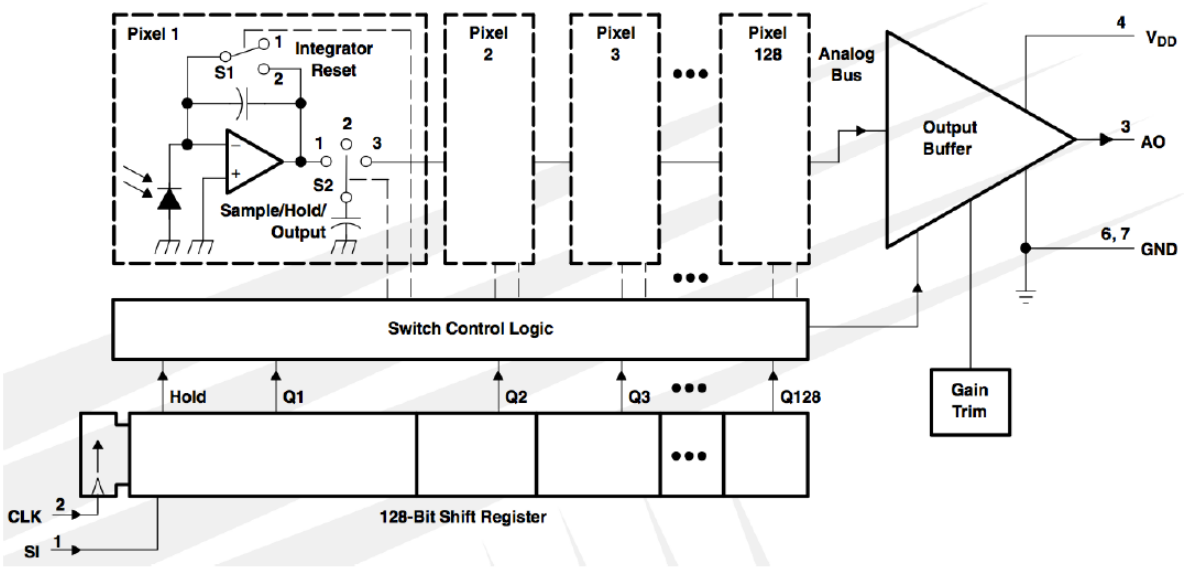


Figura 2.7 – Schema funzionale del sensore

V_{DD}: Tensione di Alimentazione
A0: Analog Output
GND: Ground
SI: Serial Input
Clk: Clock

Nello schema vediamo che l'input del sistema è la luce: essa trasporta energia e colpendo il fotodiodo genera una corrente che carica il condensatore di *sampling* attraverso il circuito di integrazione. La carica accumulatasi sul condensatore si mantiene grazie all'interruttore 2 ed è intuibile che sarà proporzionale all'intensità luminosa che ha colpito il fotodiodo e al tempo d'integrazione.

La famiglia TAOS TSL14xx deve la sua flessibilità alla possibilità di variare tale periodo di integrazione, modificando così la tensione analogica di uscita corrispondente ai vari livelli di bianco, evitando di entrare in saturazione.

Dal data sheet della telecamera troviamo anche i seguenti parametri:

- Interfaccia a 5 pin per i segnali di input e output (ground, power, SI, CLK, AO)
- Tempo di esposizione da 267µs a 68ms
- Stadio di amplificazione interno per il miglioramento del contrasto bianco/nero
- Risoluzione di 400 DPI
- Elevata linearità e uniformità
- Ampio Dynamic Range 4000:1 (72 dB)
- Tensione di uscita analogica riferita a GND
- Basso ritardo di immagine
- Frequenza di lavoro fino a 8 MHz
- Alimentazione unica a 3 o 5 V
- Nessuna resistenza di carico richiesta per il funzionamento
- Dispositivo a norma RoHS

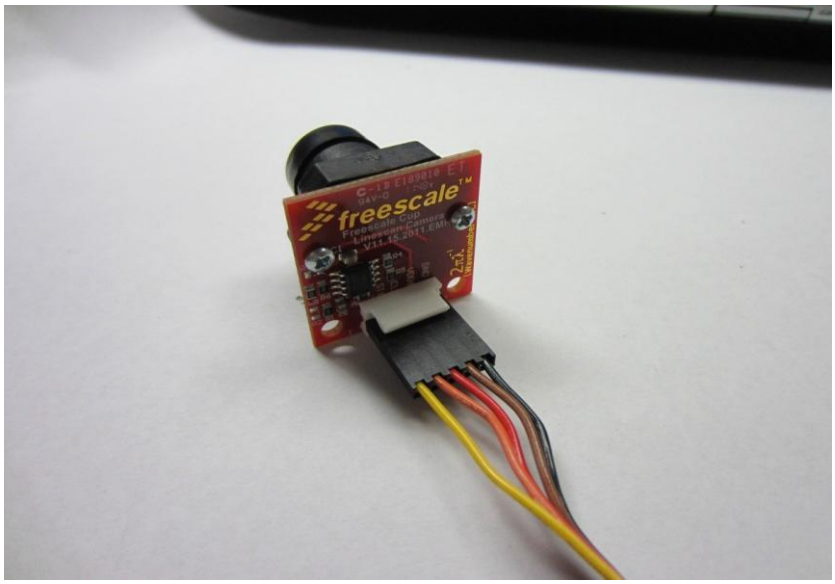


Figura 2.8 – Interfaccia a 5 pin per i segnali di I/O

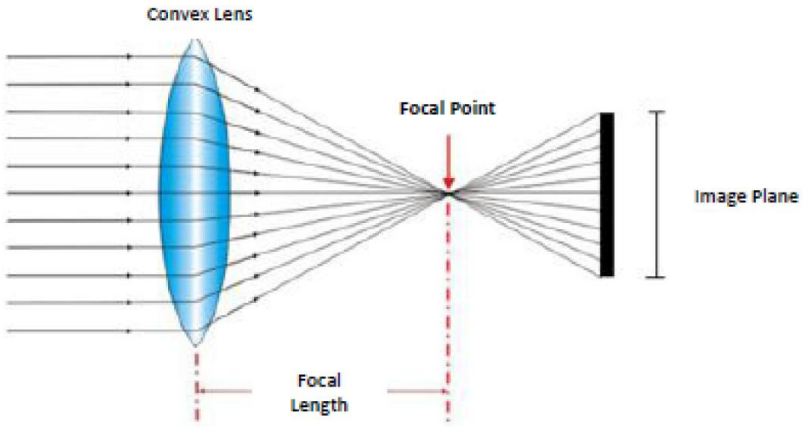


Figura 2.9 - Schematizzazione del fuoco della telecamera

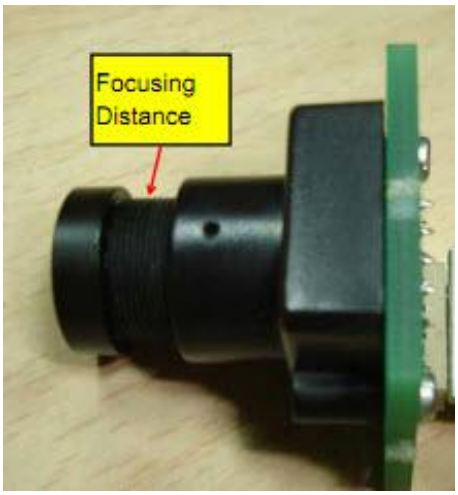


Figura 2.10 - Lente regolabile da 7.9 mm

2.2.1.1 Principali disturbi

La forma d'onda in uscita dalla telecamera assume valori di tensione elevati sui pixel che hanno ricevuto un'intensa illuminazione durante il periodo d'integrazione; in caso contrario si hanno valori di tensione più bassi. Se si rivolge la telecamera verso il terreno di gara, si nota un abbassamento della tensione di uscita proprio in corrispondenza dei pixel che puntano la linea nera. Questo avviene a causa della diversa capacità di riflettere la luce da parte della superficie nera rispetto a quella bianca. Quindi i fotodiodi, rivolti verso la superficie bianca, ricevono più luce determinando un maggiore caricamento dei condensatori di campionamento rispetto a quelli diretti sulla linea nera.
Utilizzando questo sensore si sono riscontrati quattro principali disturbi esterni che peggioravano notevolmente la precisione e l'affidabilità dei dati provenienti dalla Line Scan Camera:

1. Luminosità variabile sul retro del fotodiodo.
   Il problema qui riscontrato era la variabilità di luce nell'ambiente che oltre a dare problemi sul tracciato (poi risolti con l'introduzione di un led sul veicolo per uniformare la luce nel campo visivo della telecamera) attraversava il pcb (Printed Circuit Board) della Line Scan Camera introducendo una luce supplementare indesiderata che colpiva il sensore CMOS dal retro. Per evitare questo fenomeno, che produceva non pochi disturbi sulla misura, si è coperto il retro della sensore ottico con nastro isolante nero in modo da rendere i fotodiodi protetti dalla luce che prima li colpiva direttamente.

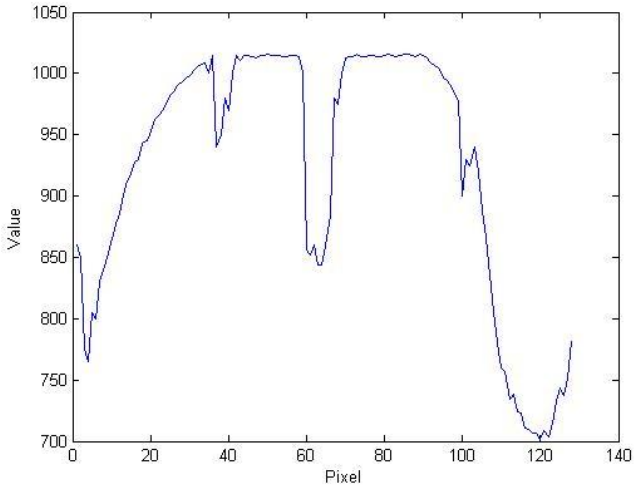


Figura 2.11 – Lettura del sensore con variabilità di luce sul retro

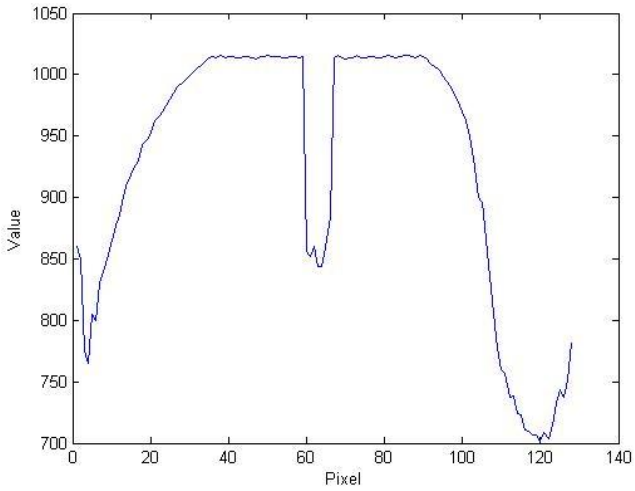


Figura 2.12 – Lettura del sensore senza variabilità di luce sul retro

2. Messa a fuoco.
   Essendo il sensore dotato di un fuoco regolabile con una lente di 7.9 mm, una volta scelti il posizionamento e l'inclinazione della telecamera è stato necessario fare una taratura del fuoco variando la distanza della lente al fine di avere valori uniformi e ben chiari nella lettura.

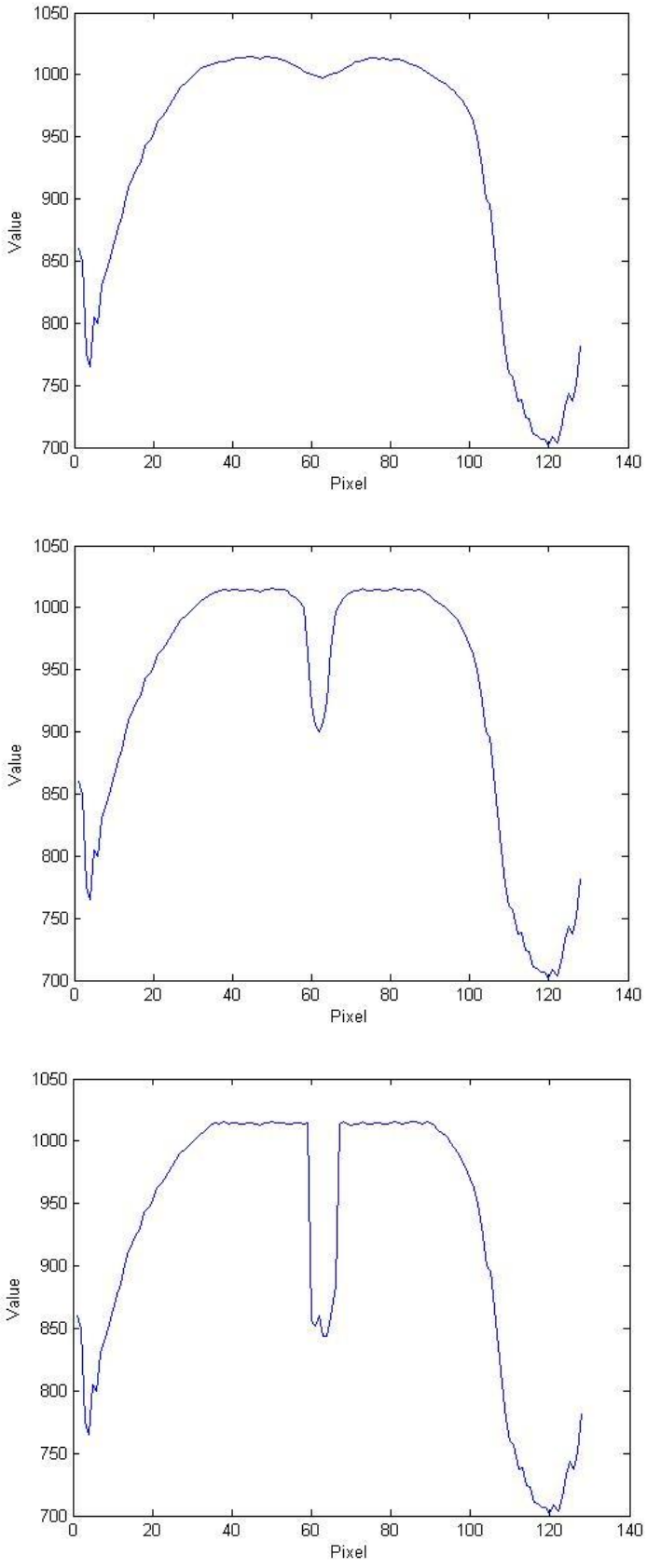


Figura 2.13 – Letture del sensore con fuoco sempre più regolato

3. *Drop off* alle estremità.
   Indipendentemente dalla posizione o da altri fattori che influenzano la lettura della Line Scan Camera era sempre presente un *drop off* ad entrambe le estremità dell'immagine dei dati come si vede nelle figure precedenti. In particolare utilizzando un metodo derivativo per il riconoscimento della linea guida (rileva le brusche variazioni dei valori dell'array della telecamera) questo risultava un problema non ininfluente, risolto poi con il restringimento del campo di visione del sensore per "tagliare" quindi i pixel disturbati.

4. Interferenza elettromagnetica.
   In molti casi i motori CC emanano interferenze causando disturbi alla telecamera e diminuendo le prestazioni del servo motore. In fase di test di velocità del veicolo infatti, oltre un determinato valore di PWM, la nitidezza del segnale proveniente dalla camera risultava inversamente proporzionale alla velocità. Questo problema è stato risolto inserendo tre capacitori da 100 nF nelle carcasse dei due motori per avere un circuito RLC (resistivo induttivo capacitivo) con effetto di filtro passa basso.

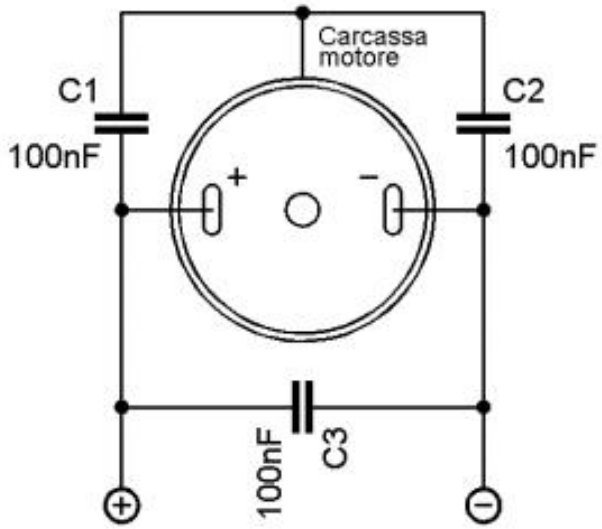


Figura 2.14 – Schematizzazione del filtro collegato al motore

Figura 2.15 – Collegamento effettivo dei filtri ai motori

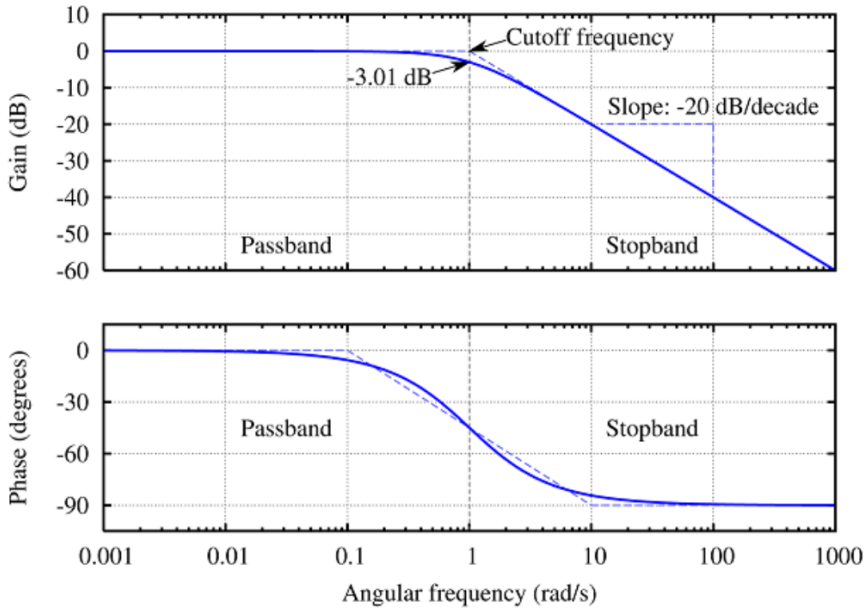


Figura 2.16 – Diagramma di Bode del filtro passa basso ideale

2.2.2 ENCODER

L'Encoder è un apparato elettromeccanico che converte la posizione angolare del suo asse rotante in un segnale elettrico. Collegato a opportuni circuiti elettronici e con appropriate connessioni meccaniche, l'encoder è in grado di misurare spostamenti angolari, movimenti rettilinei, circolari, velocità di rotazione ed accelerazioni.
Gli encoder possono essere di due tipi:

- incrementali: i segnali d'uscita sono proporzionali in modo incrementale allo spostamento effettuato rispetto ad una posizione assunta come riferimento (questi incrementi sono indipendenti dal verso di rotazione il quale non può essere rilevato da questo tipo di trasduttori).
- assoluti: ad ogni posizione dell'albero rotante corrisponde una sequenza di segmenti sul disco ben definita.

La necessità di disporre del valore istantaneo della velocità del veicolo in ogni punto del tracciato per un controllo preciso ed affidabile, ha portato ad optare per l'utilizzo di un encoder ottico. Sarebbe stato ideale averli montati sulle ruote anteriori perché non interessate da fenomeni di slittamento o bloccaggio in fase di accelerazione e frenata (causati dal collegamento dei motori al retrotreno e non all'avantreno), ma siamo stati costretti a montarli alle ruote posteriori per la facilità meccanica di installazione e per le minori sollecitazioni.
Gli encoder da noi creati (uno per ogni ruota posteriore) sono analoghi agli encoder incrementali dove però non si conta il numero di impulsi, bensì l'intervallo di tempo trascorso tra un fronte di salita e l'altro grazie alla funzionalità IPM (Input Period Measurement) dei moduli eMIOS.

Sono stati utilizzati:

- Un disco rigido di diametro 42 mm diviso in 20 spicchi regolari alternati in materiale plastico e rame:

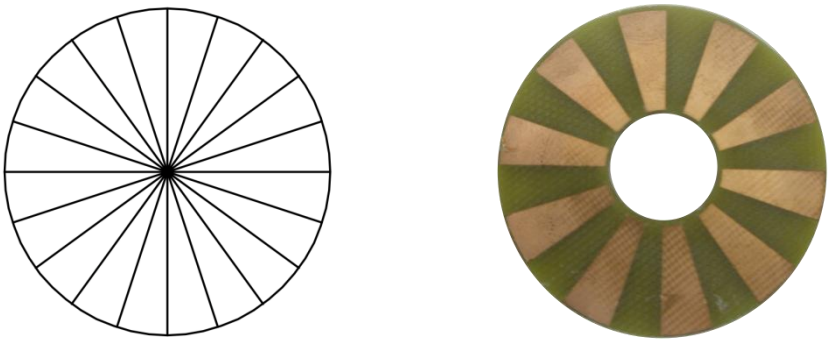


Figura 2.17 – Progetto e realizzazione del disco rigido

- Un sensore ottico riflettivo HOA1405:
  è un fotoaccoppiatore in cui vi è la combinazione, in uno stesso contenitore, di un LED e un fototransistor. Il fototransistor ha la proprietà di condurre corrente fra l'emettitore (E) e il collettore (C), proporzionale alla quantità di luce che incide sulla base. Dato che, sia l'emettitore sia il ricevitore dei raggi sono disposti sulla stessa superficie, è necessario che davanti ad entrambi sia presente una superficie riflettente, per fare in modo che il fototransistor possa ricevere i raggi che genera il LED. La superficie riflettente deve essere situata a pochi millimetri da quella su cui sono montati emettitore e ricevitore, per far si che i raggi riflessi abbiano sufficiente intensità.

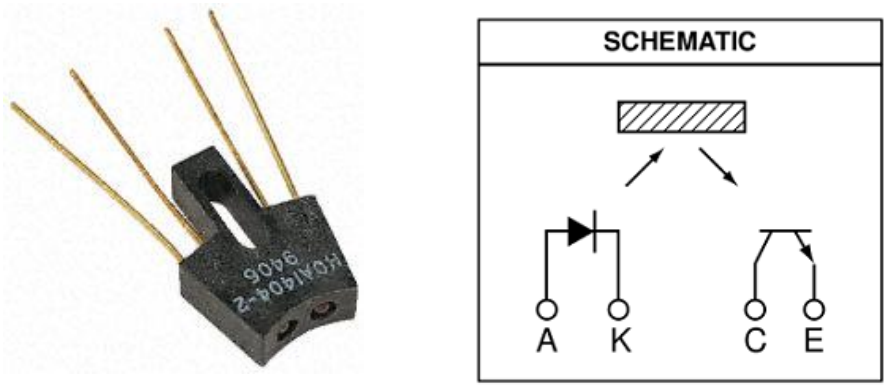


Figura 2.18 – Sensore HOA1405 e schematizzazione circuitale

- Un comparatore LM393:
  è costituito da due comparatori di tensione indipendenti progettati per operare con una singola alimentazione in un ampio intervallo di tensioni. È anche possibile lavorare con duplici alimentazioni finché la differenza tra le due vari da 2 V a 36 V e Vcc sia di almeno 1,5 V maggiore rispetto alla tensione di ingresso di modo comune. L'assorbimento di corrente è indipendente dalla tensione di alimentazione.

Figura 2.19 – Comparatore LM393 e piedinatura integrato

Il sensore riflettivo viene fissato al telaio del veicolo e fatto puntare al disco rigido calettato alla ruota posteriore come in figura 2.20.

Figura 2.20 – Montaggio encoder

Lo schema circuitale adottato è quello in figura 2.21:

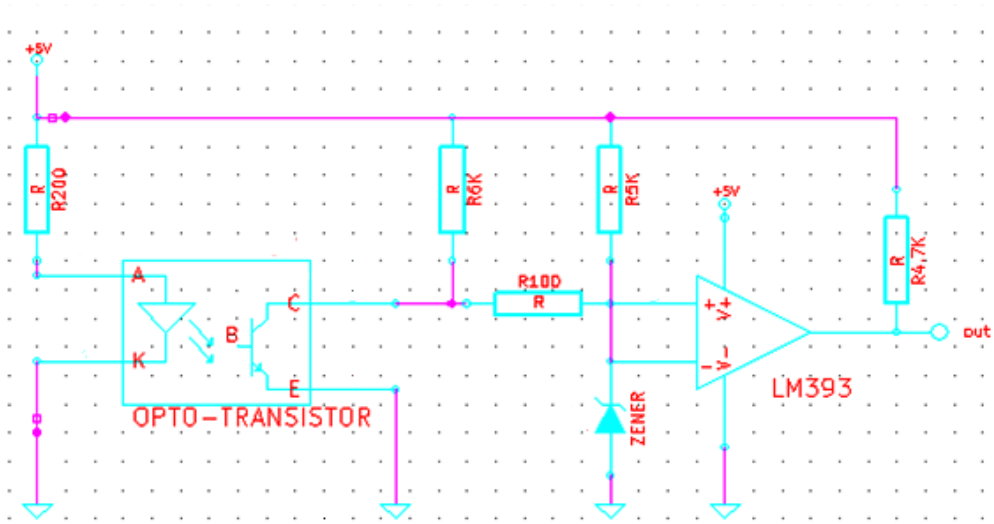


Figura 2.21 – Schema circuitale

Il segnale che si ottiene in uscita dal sensore riflettivo, quando si ha la rotazione delle ruote, risulta essere un segnale TTL (Transistor to Transistor Logic), cioè un treno di impulsi: si ha un impulso ogni qualvolta il LED illumini una zona opaca.
Il transistor ad emettitore comune, a seconda che sia illuminato o meno, lavora o in saturazione o in interdizione: il fascio luminoso, colpendo la base, crea un cortocircuito tra collettore ed emettitore connettendo così agli zero Volt il collettore. Viceversa se la base del transistor non viene irradiata il dispositivo si comporta come un circuito aperto. Si avrà allora all'uscita del sensore ottico una tensione di circa 5 V quando il transistor non viene colpito dalla luce, zero Volt altrimenti.
La successione di questi opposti stati logici in uscita dell'HOA1405 risulta però simile più ad una semisinusoide che ad un'onda quadra come illustrato in figura 2.22; proprio per questo si è deciso di introdurre l'integrato LM393 che funge da comparatore non invertente rispetto alla tensione di zener (2,2 V). In questo modo quando il segnale d'ingresso al morsetto positivo dell'operazionale è maggiore della tensione di soglia applicata a quello negativo (quando viene illuminata la regione opaca) il segnale passa di colpo a Vdd = 5V. In caso contrario (quando viene illuminata la regione riflettente) invece il valore tende a zero Volt.
A questo punto il segnale ha tempi di salita e di discesa pressoché nulli e può essere posto in ingresso ai canali eMIOS che misurano la distanza tra due fronti successivi dell'onda quadra.

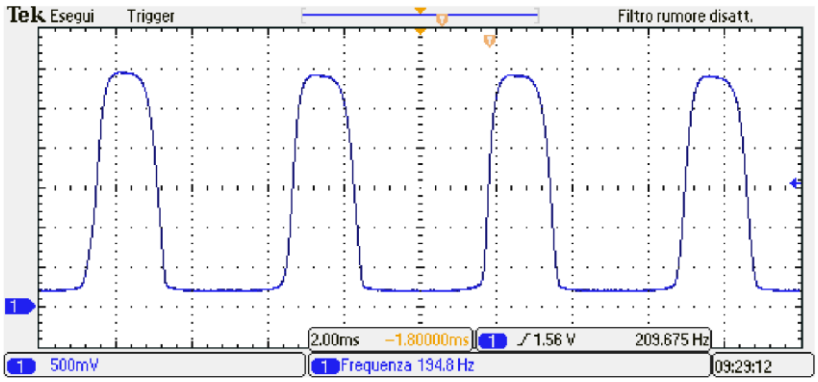


Figura 2.22 – Segnale non condizionato uscente dal sensore ottico

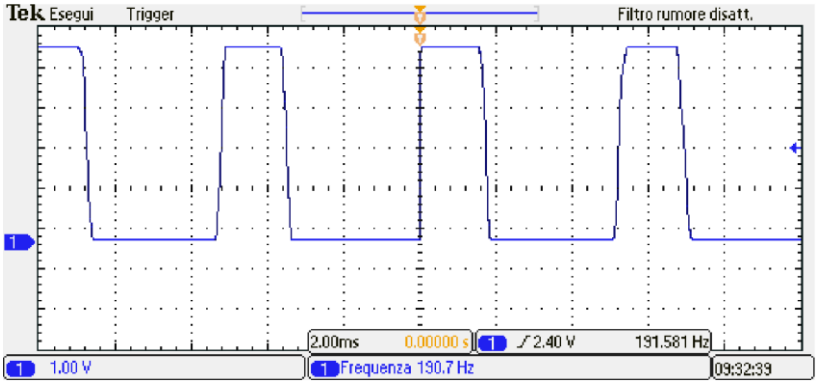


Figura 2.23 – Segnale condizionato uscente dall'operazionale

2.3 ATTUATORI

Gli attuatori sono dispositivi che convertono l'energia da una forma ad un'altra: principalmente un segnale di input elettrico in un movimento di una parte meccanica. Nel nostro caso sono il blocco motori e il servo motor.

2.3.1 MOTOR DRIVE BOARD

Lo stadio di potenza è separato dal processore, e si trova nella scheda Motor Drive, assieme all'hardware necessario ad interfacciarsi con servomotore e sensore di visione. Nella scheda sono presenti due H-bridge (ponti ad H o *chopper* a 4 quadranti) per l'alimentazione dei motori DC tramite tecnica PWM grazie alla quale è possibile variare la tensione di armatura dei motori, in modo indipendente, pur avendo a disposizione una sorgente di tensione continua a 7.2 V. È anche presente un interruttore che controlla l'alimentazione proveniente dalla batteria.

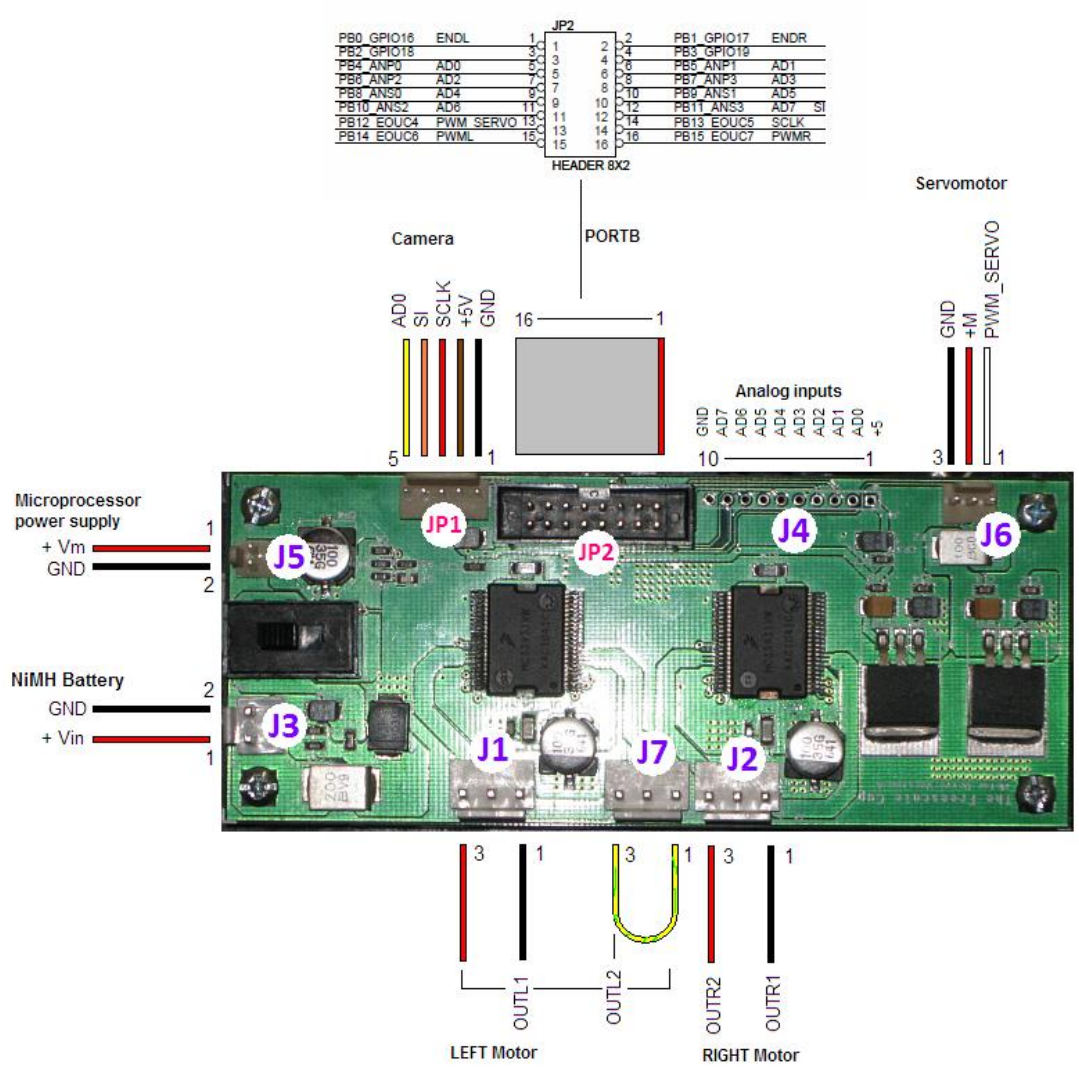


Figura 2.24 – Scheda Motor Drive con relativi segnali di I/O

I ponti ad H forniti nella scheda sono il modello MC33931: dei circuiti elettronici in grado di poter funzionare nei quattro quadranti del piano corrente-tensione sul carico. I 33931 dispongono anche di una limitazione della corrente in uscita e sono in grado di rilevare i cortocircuiti accidentali e le possibili condizioni di sovra-temperatura.
La connessione tra questa scheda ed il processore avviene tramite una piattina a 16 pin, che trasporta diversi segnali, tra cui quelli di comando per gli attuatori e quelli di lettura dei segnali provenienti dal sensore di visione.

Figura 2.25 – Ponte ad H MC33931

I due suoi piedini IN1 e IN2 sono input indipendenti e permettono il controllo separato dei 2 rami del *chopper*. Per garantire elevate prestazioni al veicolo è indispensabile la frenatura e, come si può osservare in figura 2.26, la Motor Drive non è stata predisposta per permettere ai motori di andare in retromarcia. Infatti i PIN IN1 e IN2 sono collegati alla massa della Motor Drive. Per questo motivo il PIN IN1 è stato sollevato e collegato al PIN PA6 della scheda MPC5604B, nel quale avremo in uscita o un segnale PWM oppure GND a seconda se si vuole andare in retromarcia oppure in avanti. Un procedimento analogo è stato fatto per il controllo del motore destro.

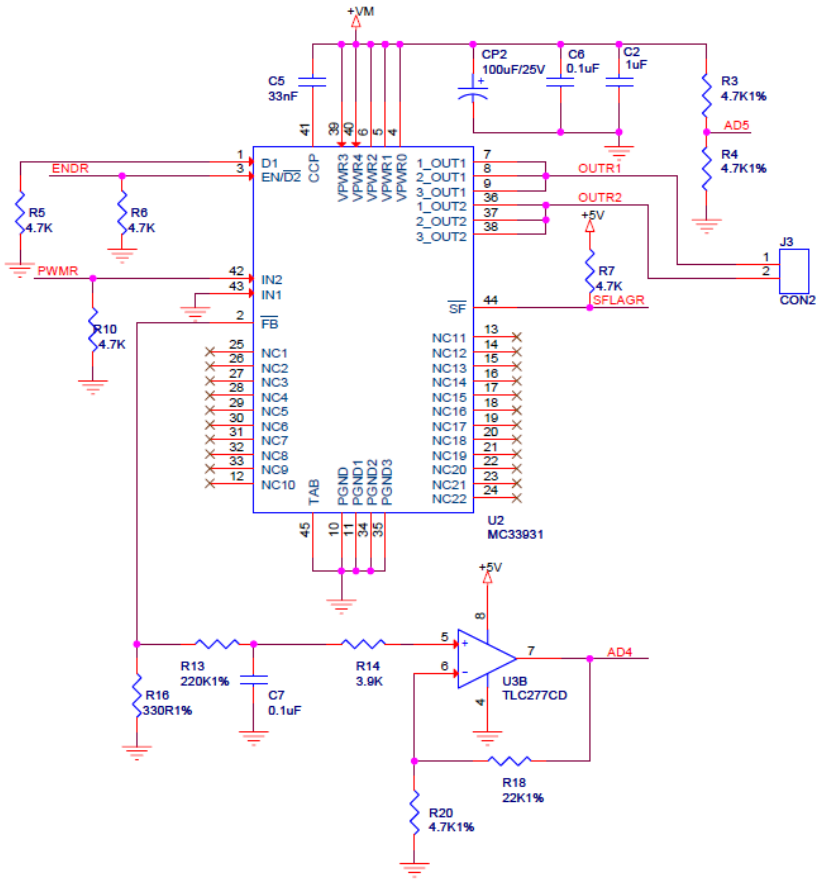


Figura 2.26 – Connessioni originali della Motor Drive

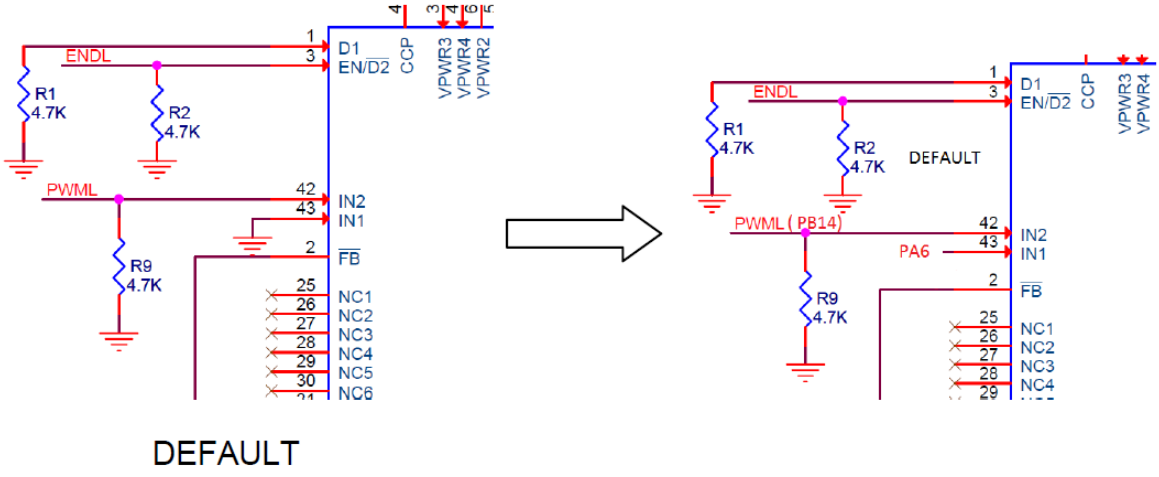


Figura 2.27 – Schematizzazione della modifica dei collegamenti di IN1 e IN2

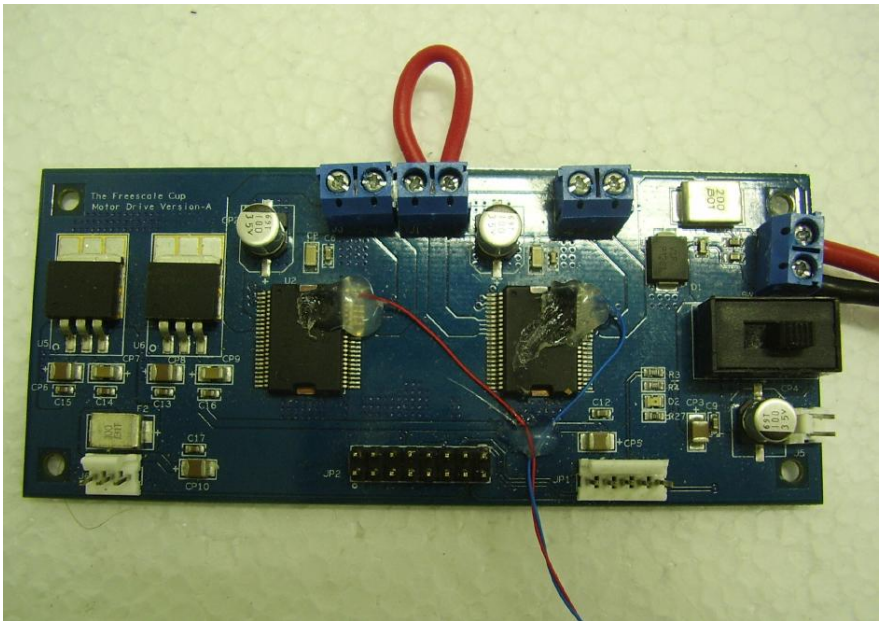


Figura 2.28 – Realizzazione del collegamento modificato di IN1 e IN2

2.3.1.1 Motori CC

Il motore elettrico è un attuatore in quanto è il componente che trasforma l'energia elettrica erogata dal convertitore statico in energia meccanica necessaria al moto delle parti meccaniche.
Sono stati forniti due motori di trazione in corrente continua del tipo Standard Motor: “rn 260 c” winding 18130 e sono stati collegati uno per ogni ruota posteriore in modo indipendente. Questo fatto è fondamentale per la progettazione del controllo del veicolo, in quanto è opportuno che un eventuale differenziale andrà programmato via software, data la mancanza di un differenziale di tipo meccanico.

Figura 2.29 – Motore in dotazione

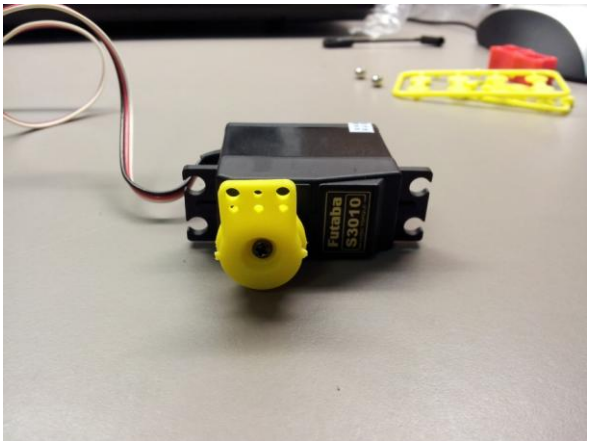
I motori in corrente continua sono caratterizzati da un sistema meccanico di spazzole e collettore per trasferire corrente al rotore. Il principale vantaggio che si è riscontrato in questo tipo di motori consiste nelle rapide accelerazioni che riescono a realizzare, mentre il maggior difetto osservato è stata una veloce e progressiva usura del sistema spazzole-collettore fortemente evidenziata dalle prestazioni.

Il catalogo dei motori riporta i seguenti dati:

Rated Voltage	V	4,5
No Load Speed	rpm	10000
No Load Current	m A	130
Max efficiency Current	m A	510
Max efficiency Speed	rpm	7950
Max efficiency Torque	gcm	18
Max output Current	m A	1070
Max output Speed	rpm	5000
Max output Torque	gcm	44
Stall current	m A	2000
Stall Torque	gcm	88

2.3.2 SERVO MOTOR

I servomotori (*servos*) sono particolari motori caratterizzati al loro interno da una periferica per l'acquisizione della posizione del rotore. Ne è stato utilizzato uno in corrente continua: il Futaba S3010 alimentato a 5V dalla scheda Motor Drive, che assorbe circa 2 A di corrente, posizionato nella parte anteriore del veicolo, collegato alle due ruote anteriori con bracci di controllo.



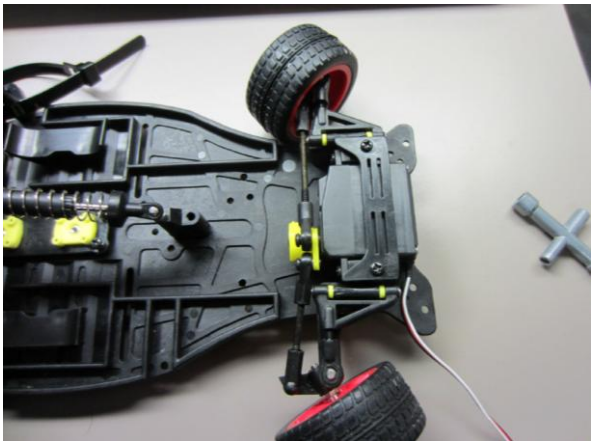


Figura 2.30 e figura 2.31 – Servomotore Futaba S3010 e suo montaggio sul veicolo

Esso trova largo impiego nella robotica e nel modellismo perché capace di produrre coppie elevate rispetto alle proprie dimensioni e di posizionarsi ad uno specifico angolo o ruotare in continuazione interfacciandosi con un microcontrollore attraverso tre fili, rosso (power), nero (ground) e bianco (control).

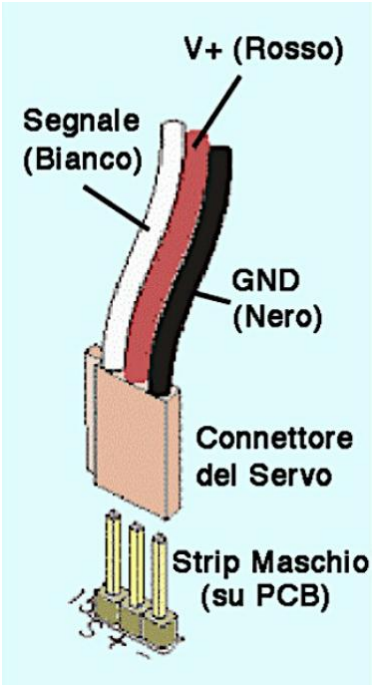


Figura 2.32 – Collegamento del servomotore

Un servo motor CC è generalmente costituito da: un motore in continua di piccole dimensioni, ma di potenza discreta, una catena di riduzione del moto, un potenziometro dedicato al feedback di posizione e da un circuito per il controllo di posizione. Inoltre nella maggior parte dei casi l'albero motore è vincolato a ruotare solo in un certo *range* di gradi, solitamente tra i 180 e i 270 gradi.
Attualmente tutti i servomotori sono di tipo proporzionale, possono quindi assumere tutte le configurazioni possibili all'interno del range di movimento. La potenza applicata al motore è proporzionale alla distanza che deve percorrere per posizionarsi correttamente. Così se l'albero necessita di spostarsi di un angolo molto ampio, esso si muoverà alla velocità massima, al contrario se necessita solo di una piccola regolazione il motore ruoterà a bassa velocità: questo tipo di controllo è detto proporzionale.
Per comunicare l'angolo a cui si desidera che l'albero si porti è necessario inviare all'azionamento un segnale attraverso il connettore di controllo. Il servo attende un impulso ogni 20 ms e l'angolo è determinato dalla durata di un impulso inviato su tale connettore, generato mediante PWM. Per tutto il tempo in cui il segnale rimane attivo in ingresso al circuito di controllo il servo manterrà la medesima posizione angolare; nel momento in cui il segnale varia il proprio periodo l'albero si riposizionerà di conseguenza.
Generalmente con un impulso di durata pari a 1.5 ms il perno del servo si posiziona esattamente al centro del suo intervallo di rotazione. Da questo punto il perno può ruotare fino a -90 gradi (senso antiorario) se l'impulso fornito ha una durata inferiore a 1.5 ms oppure fino +90 gradi (senso orario) se l'impulso fornito ha durata superiore a 1.5 ms.

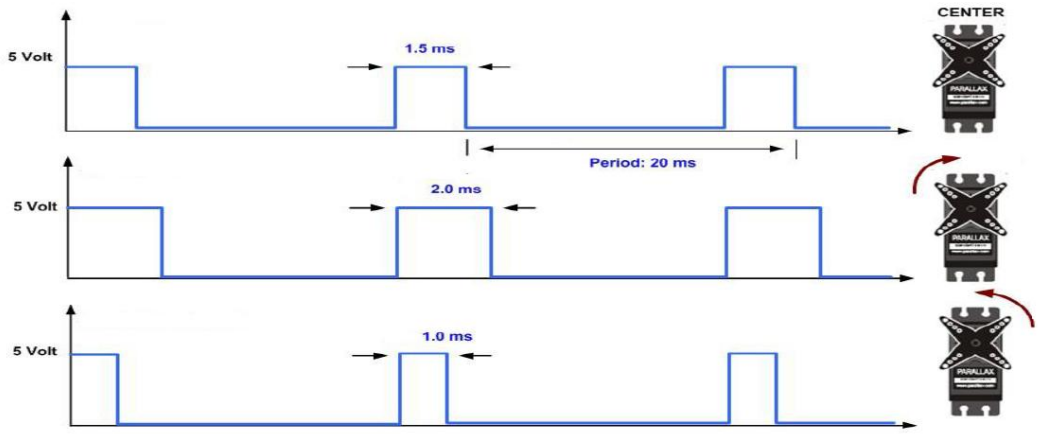


Figura 2.33 – Esempio di relazione tra durata dell'impulso e rotazione del servomotore

Inoltre dal data sheet del servomotore troviamo queste specifiche:

Basic Information

Modulation:	Analog
Torque:	**4.8V:** 72.0 oz-in (5.18 kg-cm) **6.0V:** 90.0 oz-in (6.48 kg-cm)
Speed:	**4.8V:** 0.20 sec/60° **6.0V:** 0.16 sec/60°
Weight:	1.45 oz (41.0 g)
Dimensions:	Length: 1.57 in (39.9 mm) Width: 0.79 in (20.1 mm) Height: 1.50 in (38.1 mm)
Motor Type:	3-pole
Gear Type:	Plastic
Rotation/Support:	Single Bearing

Figura 2.34 – Data sheet Servo Motor

Capitolo 3

Software

Ora che conosciamo nel dettaglio l'hardware della smart car possiamo dedicarci pienamente alla gestione e all'ottimizzazione del software, spiegando i problemi incontrati e le soluzioni adottate.
La programmazione avviene mediante il software CodeWarrior attualmente sviluppato da Freescale Semiconductor™. Questo software appartiene alla categoria degli IDE (Integrated Development Enviroment), contiene un compilatore per il linguaggio C e fornisce alcuni strumenti per il *debugging*.
Per una maggiore chiarezza nella gestione delle funzioni generate si è deciso di dividere il programma in molteplici *files* sorgenti, quali:

- telecamera.c
- pinout.c
- sterzo.c
- motori.c
- main.c

Nella programmazione, la possibilità di implementare un programma su diversi livelli permette di semplificare le operazioni di progettazione del software, di aumentare il livello di astrazione e di scomporre il problema in più parti.
Con questo criterio per utilizzare variabili dichiarate in un altro *file* sorgente come variabili globali, si è costretti a ridichiararle nel *source file* in cui si vogliono usare, oltre che con il proprio tipo, anche con la classe di memoria di tipo *extern* in modo da far capire al compilatore che si sta utilizzando la medesima variabile dichiarata in un altro *file*.

3.1 METODI AUSILIARI

Il programma che andrò ad esporre è stato scritto in linguaggio C; questo linguaggio necessita di due principali *files*: i *files* ".h" (*header files*) e quelli ".c" (*source files*). Per poter utilizzare una funzione definita altrove (es. in un altro *file* sorgente) è necessario far conoscere al compilatore il suo prototipo, ovvero quali sono, e di che tipo, i suoi parametri di input, e cosa eventualmente restituisce. Così per ogni *source* è necessario creare un *header* contenente le dichiarazioni delle funzioni usate nel rispettivo *file* ".c". Inoltre, all'inizio del *file*, bisogna includere tutti gli *headers* delle funzioni utilizzate mediante il comando *#include "header.h"*, con *header.h* il nome del *file*. Con questa direttiva di inclusione, prima della compilazione del programma, il contenuto degli *headers* viene incluso nel *file* sorgente.
Oltre a questi *header* è stato creato un altro *file* ".h": il *define.h* nel quale si inseriscono tutte le costanti usate nei vari *files* sorgenti. In questo modo se si volesse cambiare il valore di una costante usata in più *source files*, è necessario aggiornarne il valore una sola volta in questo *file*. Viene infatti utilizzato il comando *#define* che indica al preprocessore di sostituire in tutto il programma la definizione della macro con quella della stringa di sostituzione.

define.h:

```
#ifndef DEFINE_H_
#define DEFINE_H_

#define STEERING_CENTER 840                  /*Valore di centro sterzo*/
#define HALF_RANGE_STEER 335                 /*Minimo raggio di curvatura*/
#define DELTA_CALIBRATION_STEERING 15        /*Step di calibrazione sterzo*/

#define PWM_MAX_DRIVE 999                    /*Massima pwm motori*/
#define PWM_MAX_SERVO 19999                  /*Massima pwm servo*/

#define N_PIXEL 128                          /*Numero pixel telecamera*/
#define VIEW_START_LOW 5                     /*Range ristretto di lettura della*/
#define VIEW_END_LOW 123                     /*telecamera dovuto al drop off*/
#define LEN_ARRAY_CENTRO 3                   /*Lunghezza array media centilinea*/

#define MAX_ADC 1024                         /*Massimo valore ADC 10 bit*/

#define SPEED_STRAIGHT 3500                  /*Velocità rettilineo*/
#define SPEED_TURN 1950                      /*Velocità curva*/
#define DELTA_CALIBRATION_SPEED_STRAIGHT 100 /*Step di calibrazione/*
                                             /*velocità rettilineo*/
#define DELTA_CALIBRATION_SPEED_TURN 50      /*Step di calibrazione*/
                                             /*velocità curva*/

#endif /* DEFINE_H_ */
```

Il comando *#ifndef* si usa per fare in modo che il *file header* venga compilato una sola volta anche se incluso più volte in diversi *files* sorgenti: la direttiva compila il codice successivo all'istruzione solo se non è già stato definito in precedenza, altrimenti DEFINE_H_ risulterà già definito e quindi il compilatore salterà tutto il codice fino ad #endif.
Quindi l'istruzione *#ifndef* equivale a *#if 0* quando il codice è già stato definito ed è equivalente a *#if 1* quando non è ancora stato definito.

Analoga funzione del *file define.h* è quella del *delay.c*: esso contiene una serie di ritardi di varie dimensioni, generati mediante iterazioni di cicli *for*, necessari al programma principale (ad esempio come periodo di integrazione della telecamera). Richiamando queste semplici funzioni si eviterà di riscriverne tutto il corpo ogni qualvolta si vorranno utilizzare nel programma principale.

delay.c:

```
#include "MPC5604B.h"
#include "delay.h"

void Delayclock(void)
{
	int i=0;
	for(i=0;i<250;i++)
	{	}
}
void Delaylong(void)
{
	int i=0;
	for(i=0;i<20000;i++)
	{	}
}
void Delaylonglong(void)
{
	int i=0;
	for(i=0;i<10;i++)
		Delaylong();
}
void Delaycamera(void)
{
	int i=0;
	for(i=0;i<1;i++)
		Delaylong();
}
void Delaywait(void)
{
	int i=0;
	for(i=0;i<500;i++)
		Delaylong();
}
void Delayled(void)
{
	int i=0;
	for(i=0;i<5000;i++)
		Delaylong();
}
void Delay(uint16_t ms)
{
	int i=0;
	for(i=0;i<(10660 * ms);i++)
	{	}
}
void Delay_ms(uint16_t ms)
{
	int i=0;
	for(i=0;i<(10660 * ms);i++)
	{	}
}
```

3.2 TELECAMERA.C

Ricordando che la parte di tracciato bianca riesce a riflettere maggior luce rispetto a quella nera (relativa alla linea), si avranno pixel della telecamera con una componente di energia luminosa diversa a seconda della parte di tracciato a cui puntano.
Una "pecca" della Line Scan Camera è l'uscita di tipo analogico, quindi è fondamentale adottare un metodo semplice e rapido per processare il segnale.
Il metodo derivativo che si è adottato per il rilevamento della linea guida risulta molto efficace in quanto permette di tollerare senza problemi il rumore luminoso di fondo (perché caratterizzato da variazioni di luminosità non molto brusche), e nello stesso tempo di essere notevolmente sensibile alle decise variazioni come quelle causate dalla visione di una linea nera su sfondo bianco.

```
#include "MPC5604B.h"
#include "telecamere.h"
#include "delay.h"
#include "define.h"

uint16_t cameraLow[N_PIXEL];
int16_t DevArray[127];
uint16_t Soglia;
int16_t SogliaMin = 0;
int16_t SogliaMax = 0;
uint8_t TrovatoStop = 0;
```

La funzione *void CAMERA(*camera)* esegue la lettura dell'uscita della telecamera e provvede alla generazione dei segnali di *clock* e *serial input*. La generazione dei segnali CK e SI avviene attraverso l'abilitazione e l'attivazione/disattivazione delle porte PCR[27] e PCR [29] (rispettivamente PB[11] e PB[13]); si nota come i due segnali siano sfasati di un *delay* pari a mezzo periodo di oscillazione in modo da mettere in fase CK ed A0, come richiesto dal data sheet della camera. Il parametro passato è un array d'interi a 16 bit unsigned, che possono rappresentare valori da zero a 2^{16} (65536). I valori in uscita dall'ADC (Analog Digital Converter) sono salvati nell'array *camera*. L'ADC interno alla scheda, ha 10 bit e quindi è in grado di restituire valori tra zero e 1024. La risoluzione dell'ADC è definita come il rapporto tra l'*input range* del dispositivo e il numero dei livelli di quantizzazione disponibili. Nel caso in esame si ha:

$$\delta = \frac{5\ \mathrm{V}}{2^{10}} = 4.88\ \mathrm{mV}.$$

Dal momento che la telecamera è in grado di leggere la linea molto velocemente, mentre il servo motore può aggiornare la sua posizione solo ogni 20 ms, ci sono più letture della telecamera prima che il servo si aggiorni. Per rendere la sua variazione meno scattosa (evitando dunque di far perdere più facilmente aderenza in curva alla smart car) e anche per rendere il veicolo meno influenzato da eventuali centri linea errati causati da letture sbagliate, si è deciso di impostare il servo non più al valore dell'ultimo centro linea trovato, ma alla media di questo con i due centri linea trovati in precedenza. Si è quindi creato un array di tipo FIFO (First In First Out) per memorizzare i tre centri linea su cui fare la media che costituisce una sorta di filtro.
Per la completa comprensione del programma si ricorda che è buona regola "forzare" esplicitamente le conversioni tramite l'operatore unario di *cast*, anche se la conversione è automatica nel caso si voglia trasformare un operando "più piccolo" in uno "più grande" senza perdita di informazioni. Inoltre è necessario sapere che un puntatore è una variabile che contiene l'indirizzo di una locazione di memoria: l'operatore di deriferimento * restituisce il valore di tale variabile che si trova nell'indirizzo indicato nell'argomento che lo segue, mentre l'operatore di indirizzo & significa "indirizzo della variabile" che lo segue.

```
/*Acquisizione dell'array della telecamera*/
void CAMERA(uint16_t* camera)
{
	uint8_t i = 0;
	uint8_t j = 0;

	SIU.PCR[27].R = 0x0200;   /* Program the Sensor read start pin as output*/
	SIU.PCR[29].R = 0x0200;   /* Program the Sensor Clock pin as output*/
	SIU.PGPDO[0].R &= ~0x000000014;        /* All port line low */
	SIU.PGPDO[0].R |= 0x00000010;          /* Sensor read start High */
	Delayclock();
	SIU.PGPDO[0].R |= 0x00000004;          /* Sensor Clock High */
	Delayclock();
	SIU.PGPDO[0].R &= ~0x00000010;         /* Sensor read start Low */
	Delayclock();
	SIU.PGPDO[0].R &= ~0x00000004;         /* Sensor Clock Low */
	Delayclock();
	for (i = 0; i < N_PIXEL; i++)
	{
		Delayclock();
		SIU.PGPDO[0].R |= 0x00000004;    /* Sensor Clock High */
		ADC.MCR.B.NSTART=1;         /* Trigger normal conversions for ADC0 */
		while (ADC.MCR.B.NSTART == 1) {};
		*(camera + i) = (uint16_t)(ADC.CDR[1].B.CDATA);/*put the value from*/
                                                /*ADC to the array with cast*/
		Delayclock();
		SIU.PGPDO[0].R &= ~0x00000004;         /* Sensor Clock Low */
	}
	Delaycamera();
}
```

L'ultimo dei *delay* introdotti rappresenta il tempo d'integrazione della telecamera, cioè il periodo in cui la corrente che si genera carica i condensatori connessi ai fotodiodi.

Il metodo *void Derivata (Array, DevArray)* svolge una sorta di derivata confrontando pixel adiacenti della telecamera (presenti nell'array **Array*), eseguendone la differenza e salvando questo risultato nell'array **DevArray*.

```
/*Derivata array telecamera*/
void Derivata(uint16_t* Array, int16_t* DevArray)
{
	int8_t ix;
	int8_t iy;
	int16_t work;
	int8_t m;
	int16_t tmp2;
	ix = 1;
	iy = 0;
	 work = (int16_t)*(Array);
	 for (m=0; m<127; m++)
	{
		tmp2 = work;
		work = (int16_t)*(Array + ix);
		tmp2 = (int16_t)*(Array + ix) - tmp2;
		ix++;
		*(DevArray + iy) = tmp2;
		iy++;
	}
}
```

Si osserva che i bordi della linea nera coincidono con i punti a derivata più elevata in valore assoluto, mentre il rumore di sottofondo ha valori molto più esigui.

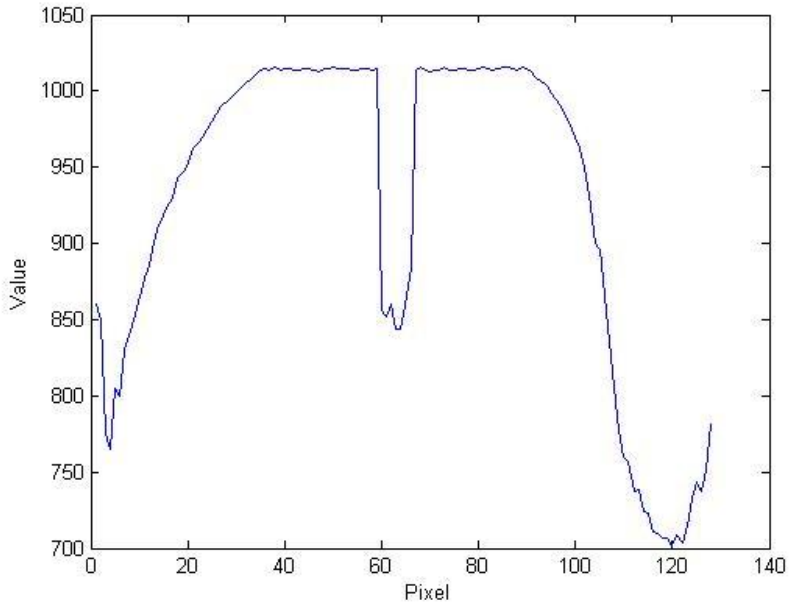


Figura 3.1 – Immagine dei valori dell'array

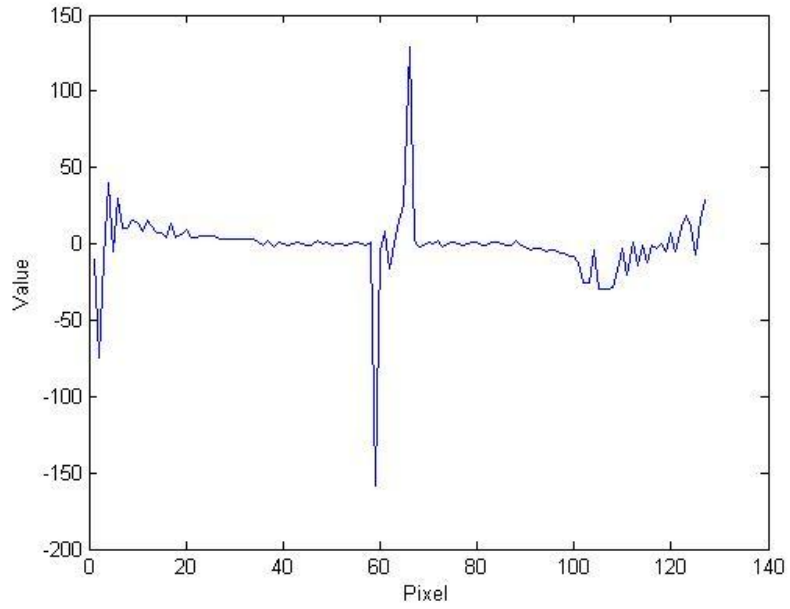


Figura 3.2 – Immagine dei valori delle derivate

La funzione *void FoundCenter(*Array, *lineStartSxDx_presunta, *lineEndSxDx_presunta)* è un metodo ausiliario alla funzione *uint8_t FIND_CENTER_LINE (*Array, lineCenterPrev)*. Infatti questo metodo svolge solamente una scansione dell'array delle derivate, eliminando il *drop off* alle estremità, trovandone il valore minimo e massimo e salvando le loro posizioni rispettivamente in**lineStartSxDx_presunta* e **lineEndSxDx_presunta*.

```
/*Trova min e max derivata in lineStart e lineEnd*/
void FoundCenter(int16_t* Array,uint16_t* lineStartSxDx_presunta ,uint16_t*
lineEndSxDx_presunta)
{
	uint16_t j = 0;
	SogliaMin = 0;
	SogliaMax = 0;
	j = VIEW_START_LOW;
	while(j<VIEW_END_LOW)
	{
		if(*(Array + j) < SogliaMin)
		{
			*lineStartSxDx_presunta = j;
			SogliaMin = *(Array + j);
		}
		if(*(Array + j) > SogliaMax)
		{
			*lineEndSxDx_presunta = j;
			SogliaMax = *(Array + j);
		}
		j++;
	}
}
```

Il metodo *uint8_t FIND_CENTER_LINE (*Array, lineCenterPrev)* è il programma principale della telecamera: quello che riesce ad individuare la linea nera sullo sfondo bianco richiamando i metodi ausiliari *Derivata (Array, DevArray)* e *FoundCenter(*Array, *lineStartSxDx_presunta, *lineEndSxDx_presunta)* e verificando che la linea trovata soddisfi certi criteri di sicurezza in termini di larghezza. Se l'algoritmo non riuscisse a rilevare la linea nera, si imposterebbe come riferimento del centro linea trovato, o il primo o l'ultimo pixel della telecamera a seconda che quello precedente sia stato trovato nella prima o nella seconda metà dell'array.

```
/*Calcolo del centro linea dato un array da una telecamera*/
uint8_t FIND_CENTER_LINE(uint16_t* Array, uint8_t lineCenterPrev)
{

	uint16_t   lineStartSxDx = 0;
	uint16_t   lineEndSxDx = 0;
	uint8_t    lineCenter;

	Derivata(Array,DevArray);
	FoundCenter(DevArray, &lineStartSxDx, &lineEndSxDx);
	if(!TrovatoStop)
		TrovatoStop = FindStop(DevArray, &lineStartSxDx, &lineEndSxDx);
	if ((lineStartSxDx != 0) && (lineEndSxDx != 0) && (lineEndSxDx –
	     lineStartSxDx < 20) && (lineEndSxDx - lineStartSxDx > 3))
	{
		lineCenter = (uint8_t)(( lineStartSxDx + lineEndSxDx ) / 2);
	}
	else
	{
		if (lineCenterPrev > 64)
			lineCenter = 128;
		else
			lineCenter = 0;
	}
	return lineCenter;
}
```

Tutto questo continua ad essere iterato anche nel caso si abbia riconosciuto il segnale di stop, mediante il metodo *uint8_t FindStop (*Array, *lineStartSxDx_presunta, *lineEndSxDx_presunta)*, in modo da continuare a seguire la linea anche in fase di frenatura e di fermata. Questa funzione cerca nell'array della derivata gli altri due picchi più esterni (uno positivo e uno negativo) relativi al segnale di stop con lo stesso principio con il quale si erano trovati quelli relativi alla linea guida nera. Dopo averli riconosciuti, ed aver verificato ancora delle diverse condizioni di sicurezza sulla distanza di questi picchi, il metodo restituisce "1" se il segnale di stop è stato rilevato, "0" altrimenti. Questo valore restituito verrà successivamente utilizzato per fermare o meno il veicolo.

```
uint8_t FindStop (int16_t* Array ,uint16_t* lineStartSxDx_presunta, uint16_t*
lineEndSxDx_presunta)
{
	uint16_t j = VIEW_START_LOW;
	uint16_t lineStartSxDx_seconda = 0;
	uint16_t lineEndSxDx_seconda = 0;
	uint8_t DistanzaStop = 0;

	Soglia = (uint16_t)(SogliaMax - 30);
	while(j<(*lineEndSxDx_presunta-10))
	{
		if((*(Array + j) > Soglia))
			lineStartSxDx_seconda = j;
		j++;
	}
	j = VIEW_END_LOW;
	while(j>(*lineStartSxDx_presunta+10))
	{
		if((*(Array + j) < -Soglia))
			lineEndSxDx_seconda = j;
		j--;
	}
	DistanzaStop = (uint8_t)(lineEndSxDx_seconda - lineStartSxDx_seconda);
	if ((lineStartSxDx_seconda!=0) && (lineEndSxDx_seconda!=0))
	{
		if((DistanzaStop > 25) && (DistanzaStop < 35))
			return 1;
		else
			return 0;
	}
	else
		return 0;
}
```

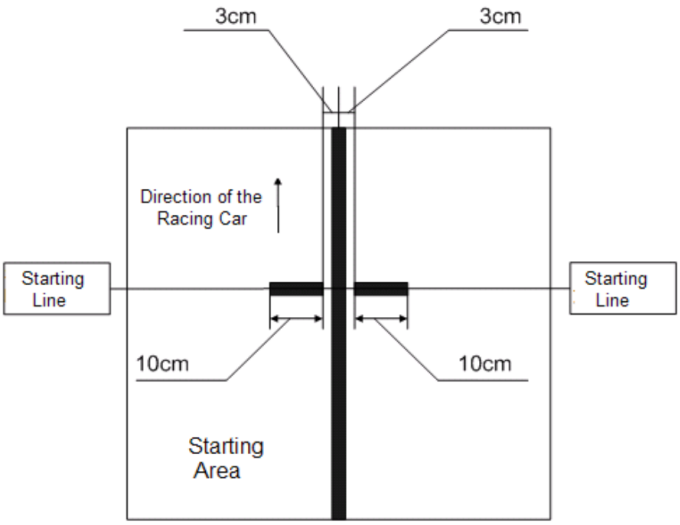


Figura 3.3 – Segnale di stop

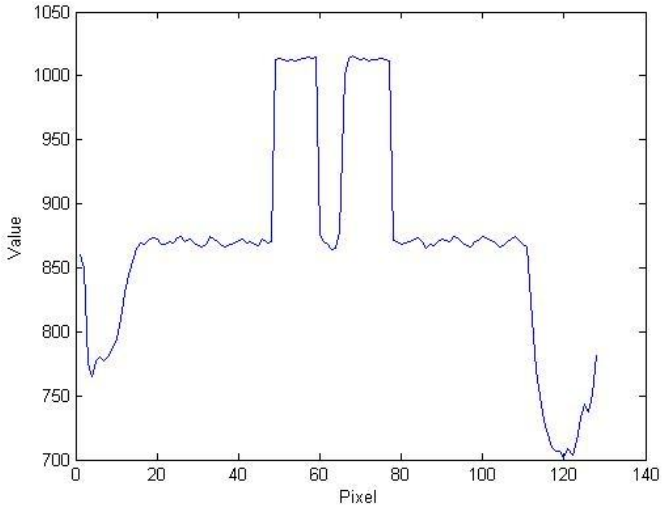


Figura 3.4 – Array telecamera

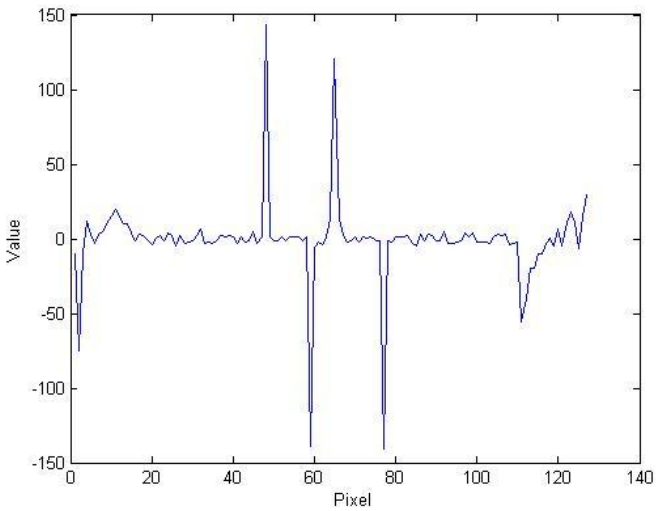


Figura 3.5 – Array derivate

3.3 PINOUT.C

Le funzioni appartenenti a questo *file* sono di basso livello di astrazione in quanto sono scritte in linguaggio macchina per interfacciarsi direttamente con le linee del controller, i moduli e le porte.

```
#include "MPC5604B.h"
#include "pinout.h"
#include "define.h"
```

Qui si inizializzano le porte del microcontrollore associandole ai canali di precisione dell'ADC.

```
void initPads (void)
{
      SIU.PCR[20].R = 0x2000;          /* Camera: attiva PB[4] as ANP0 */
      SIU.PCR[21].R = 0x2000;          /* Camera: attiva PB[5] as ANP1 */
      /*Motori*/
      SIU.PCR[16].R = 0x0200;          /* Program the drive enable pin of*/
                                       /* Left Motor as output*/
      SIU.PCR[17].R = 0x0200;          /* Program the drive enable pin of*/
                                       /* Right Motor as output*/
      SIU.PGPDO[0].R = 0x0000C000;     /* Enable Both the motors */
}
```

Si analizzano ora nel dettaglio le istruzioni in linguaggio C necessarie ad abilitare ed inizializzare l'ADC per acquisire un campione di tensione analogico in ingresso da un pin della scheda madre.
Innanzitutto è necessario inizializzare la modalità di acquisizione dei campioni tramite ADC. Una volta abilitati i canali di ingresso è possibile scegliere tra le modalità *one shot* e *scan*. In questo caso impostando a zero il bit MODE nel MCR (Main Configuration Register) si abilita la modalità *one shot*.

```
void initADC(void)
{
   ADC.MCR.R = 0x00000000;       /* Initialize ADC one shot mode*/
   ADC.NCMR[0].R = 0x0000FFFF;   /* Select ANP1:16 inputs for normal conversion */
   ADC.NCMR[1].R = 0x0000FFFF;   /* Select ANS1:16 inputs for normal conversion */
   ADC.CTR[0].R = 0x00008606;    /* Conversion times for 32MHz ADClock */
}
```

Dopo aver abilitato la generazione dei clock di sistema si attivano i moduli eMIOS_0 e eMIOS_1 impostando il *prescaler* in modo da generare un clock interno di 1 MHz grazie alla divisione per 64 del clock di sistema a 64 MHz:

```
void initEMIOS_0(void)
{
   EMIOS_0.MCR.B.GPRE= 63  /* Divide 64 MHz sysclk by 63+1=64 for 1MHz eMIOS clk*/
   EMIOS_0.MCR.B.GPREN = 1;      /* Enable eMIOS clock */
   EMIOS_0.MCR.B.GTBE = 1        /* Enable global time base */
   EMIOS_0.MCR.B.FRZ = 1;        /* Enable stopping channels when in debug mode */
}


void initEMIOS_1(void)
{
   EMIOS_1.MCR.B.GPRE= 63; /* Divide 64 MHz sysclk by 63+1=64 for 1MHz eMIOS clk*/
   EMIOS_1.MCR.B.GPREN = 1;      /* Enable eMIOS clock */
```

```
   EMIOS_1.MCR.B.GTBE = 1;       /* Enable global time base */
   EMIOS_1.MCR.B.FRZ = 1;        /* Enable stopping channels when in debug mode */
}
```

Ora si configurano i vari canali eMIOS utilizzati. Il canale 8 dell'eMIOS_1 è il contatore relativo agli encoder che lo usano nei canali 11 e 12; il canale 23 dell'eMIOS_0 è il contatore generale interno di questo modulo usato dai canali 0 e 4 per il servo motor e 2, 3, 6 e 7 per i motori CC.

```
void initEMIOS_1ch8(void)              /* EMIOS 0 CH 0: Modulus Up Counter */
{
   EMIOS_1.CH[8].CADR.R = 59999;/* Period will be 59999+1=60000 clocks (60 msec)*/
   EMIOS_1.CH[8].CCR.B.MODE = 0x50;/* Modulus Counter Buffered (MCB) */
   EMIOS_1.CH[8].CCR.B.BSL = 0x3;  /* Use internal counter */
   EMIOS_1.CH[8].CCR.B.UCPRE=0;    /* Set channel prescaler to divide by 1 */
   EMIOS_1.CH[8].CCR.B.UCPEN = 1;  /* Enable prescaler; uses default divide by 1*/
   EMIOS_1.CH[8].CCR.B.FREN = 1;   /* Freeze channel counting when in debug mode*/
}


Void initEMIOS_0ch23(void)             /* EMIOS 0 CH 23: Modulus Up Counter */
{
   EMIOS_0.CH[23].CADR.R = PWM_MAX_DRIVE;    /* Period will be 999+1=1000 clocks*/
                                             /*(1 msec)*/
   EMIOS_0.CH[23].CCR.B.MODE = 0x50;   /* Modulus Counter Buffered (MCB) */
   EMIOS_0.CH[23].CCR.B.BSL = 0x3;     /* Use internal counter */
   EMIOS_0.CH[23].CCR.B.UCPRE=0;       /* Set channel prescaler to divide by 1 */
   EMIOS_0.CH[23].CCR.B.UCPEN = 1; /* Enable prescaler; uses default divide by 1*/
   EMIOS_0.CH[23].CCR.B.FREN = 1;  /* Freeze channel counting when in debug mode*/
}


void initEMIOS_0ch0(void)              /* EMIOS 0 CH 0: Modulus Up Counter */
{
   EMIOS_0.CH[0].CADR.R = PWM_MAX_SERVO; /* Period will be 19999+1=20000 clocks/*
                                          /*(20 msec)*/
   EMIOS_0.CH[0].CCR.B.MODE = 0x50;    /* Modulus Counter Buffered (MCB) */
   EMIOS_0.CH[0].CCR.B.BSL = 0x3;      /* Use internal counter */
   EMIOS_0.CH[0].CCR.B.UCPRE=0;        /* Set channel prescaler to divide by 1 */
   EMIOS_0.CH[0].CCR.B.UCPEN = 1;  /* Enable prescaler; uses default divide by 1*/
   EMIOS_0.CH[0].CCR.B.FREN = 1;   /* Freeze channel counting when in debug mode*/
}


/*CH2 = Motor Reverse Left*/
void initEMIOS_0ch2(void)         /* EMIOS 0 CH 2: Output Pulse Width Modulation*/
{
   EMIOS_0.CH[2].CADR.R = 0;     /* Leading edge when channel counter bus=0*/
   EMIOS_0.CH[2].CBDR.R = 0;     /* Trailing edge when channel counter bus=1000*/
   EMIOS_0.CH[2].CCR.B.BSL = 0x0;      /* Use counter bus A (default) */
   EMIOS_0.CH[2].CCR.B.EDPOL = 1;      /* Polarity-leading edge sets output */
   EMIOS_0.CH[2].CCR.B.MODE = 0x60;    /* Mode is OPWM Buffered */
   SIU.PCR[2].R = 0x0600;              /* MPC56xxS: Assign EMIOS_0 ch 2 to pad */
}


/*CH3 = Motor Reverse Right*/
void initEMIOS_0ch3(void)         /* EMIOS 0 CH 3: Output Pulse Width Modulation*/
{
   EMIOS_0.CH[3].CADR.R = 0;     /* Leading edge when channel counter bus=0*/
   EMIOS_0.CH[3].CBDR.R = 0;     /* Trailing edge when channel's counter bus=1000*/
```

```
   EMIOS_0.CH[3].CCR.B.BSL = 0x0;      /* Use counter bus A (default) */
   EMIOS_0.CH[3].CCR.B.EDPOL = 1;      /* Polarity-leading edge sets output*/
   EMIOS_0.CH[3].CCR.B.MODE = 0x60;    /* Mode is OPWM Buffered */
   SIU.PCR[3].R = 0x0600;              /* MPC56xxS: Assign EMIOS_0 ch 3 to pad */
}


/*CH4 = Servo Motor*/
void initEMIOS_0ch4(void)       /* EMIOS 0 CH 4: Output Pulse Width Modulation*/
{
   EMIOS_0.CH[4].CADR.R = 0;     /* Leading edge when channel counter bus=0*/
   EMIOS_0.CH[4].CBDR.R = STEERING_CENTER;   /* Trailing edge when cannel*/
                                             /*counter bus = */
                                             /*STEERING_CENTER Middle,*/
                                             /*+HALF_RANGE_STEER Right Max, */
                                             /*-HALF_RANGE_STEER Left Max*/
   EMIOS_0.CH[4].CCR.B.BSL = 0x01;     /* Use counter bus B */
   EMIOS_0.CH[4].CCR.B.EDPOL = 1;      /* Polarity-leading edge sets output */
   EMIOS_0.CH[4].CCR.B.MODE = 0x60;    /* Mode is OPWM Buffered */
   SIU.PCR[28].R = 0x0600;             /* MPC56xxS: Assign EMIOS_0 ch 6 to pad */
}


/*CH6 = Motor Forward Left*/
void initEMIOS_0ch6(void)       /* EMIOS 0 CH 6: Output Pulse Width Modulation*/
{
   EMIOS_0.CH[6].CADR.R = 0;     /* Leading edge when channel counter bus=0*/
   EMIOS_0.CH[6].CBDR.R = 0;     /* Trailing edge when channel counter bus=1000*/
   EMIOS_0.CH[6].CCR.B.BSL = 0x0;      /* Use counter bus A (default) */
   EMIOS_0.CH[6].CCR.B.EDPOL = 1;      /* Polarity-leading edge sets output */
   EMIOS_0.CH[6].CCR.B.MODE = 0x60;    /* Mode is OPWM Buffered */
   SIU.PCR[30].R = 0x0600;             /* MPC56xxS: Assign EMIOS_0 ch 6 to pad */
}


/*CH7 = Motor Forward Right*/
void initEMIOS_0ch7(void)       /* EMIOS 0 CH 7: Output Pulse Width Modulation*/
{
   EMIOS_0.CH[7].CADR.R = 0;     /* Leading edge when channel counter bus=0*/
   EMIOS_0.CH[7].CBDR.R = 0;     /* Trailing edge when channel's counter bus=1000*/
   EMIOS_0.CH[7].CCR.B.BSL = 0x0;      /* Use counter bus A (default) */
   EMIOS_0.CH[7].CCR.B.EDPOL = 1;      /* Polarity-leading edge sets output*/
   EMIOS_0.CH[7].CCR.B.MODE = 0x60;    /* Mode is OPWM Buffered */
   SIU.PCR[31].R = 0x0600;             /* MPC56xxS: Assign EMIOS_0 ch 7 to pad */
}


void initEMIOS_1ch11(void)
{
   EMIOS_1.CH[11].CCR.B.BSL = 1;     /* Use counter bus A which is eMIOS Ch 23 */
   EMIOS_1.CH[11].CCR.B.EDPOL = 0;   /* Edge Select- Single edge trigger (count) */
   EMIOS_1.CH[11].CCR.B.EDSEL = 0;   /* Edge Select- Single edge trigger (count) */
   EMIOS_1.CH[11].CCR.B.FCK = 0;     /* Input filter will use main clock */
   EMIOS_1.CH[11].CCR.B.UCPRE = 3;
   EMIOS_1.CH[11].CCR.B.UCPEN = 1;
   EMIOS_1.CH[11].CCR.B.IF = 4;      /* Input filter uses 8 clock periods */
   EMIOS_1.CH[11].CCR.B.FEN = 1;     /* Enable Interrupt */
   EMIOS_1.CH[11].CCR.B.MODE = 5;    /* Mode is PEC, continuous */
   SIU.PCR[98].B.PA = 3;               /* Initialize pad for eMIOS channel */
   SIU.PCR[98].B.IBE = 1;              /* Initialize pad for input */
}
```

```
void initEMIOS_1ch12(void)
{
   EMIOS_1.CH[12].CCR.B.BSL = 1;     /* Use counter bus A which is eMIOS Ch 23 */
   EMIOS_1.CH[12].CCR.B.EDPOL = 0;   /* Edge Select- Single edge trigger (count) */
   EMIOS_1.CH[12].CCR.B.EDSEL = 0;   /* Edge Select- Single edge trigger (count) */
   EMIOS_1.CH[12].CCR.B.FCK = 0;     /* Input filter will use main clock */
   EMIOS_1.CH[12].CCR.B.UCPRE = 3;
   EMIOS_1.CH[12].CCR.B.UCPEN = 1;
   EMIOS_1.CH[12].CCR.B.IF = 4;        /* Input filter uses 8 clock periods */
   EMIOS_1.CH[12].CCR.B.FEN = 1;       /* Enable Interrupt */
   EMIOS_1.CH[12].CCR.B.MODE = 5;      /* Mode is PEC, continuous */
   SIU.PCR[99].B.PA = 3;               /* Initialize pad for eMIOS channel */
   SIU.PCR[99].B.IBE = 1;              /* Initialize pad for input */
}
```

3.4 STERZO.C

Il seguente algoritmo serve per calibrare e direzionare la sterzata del veicolo in funzione dei parametri acquisiti con i sistemi di visione.
La differenza tra il riferimento e la posizione corrente genera l'errore di posizione, che denota il grado di sfasamento tra la direzione di marcia del veicolo e quella della linea. L'errore creatosi dev'essere opportunamente compensato per evitare il deragliamento della macchina. Questo è il compito del controllore di sterzo di tipo PID.
L'uscita del controllore è una tensione di tipo PWM necessaria per comandare il servo motore di sterzo, il quale provvede al posizionamento del proprio albero e conseguentemente alla sterzata delle ruote. Il comando di sterzo provoca una variazione della posizione del veicolo rispetto alla linea che è rilevata dal sistema di acquisizione (camera + stimatore), generando un nuovo errore di posizione da compensare. Il sistema di controllo che si viene a creare sfrutta quindi i vantaggi offerti dalla retroazione, in termini di affidabilità e controllabilità.

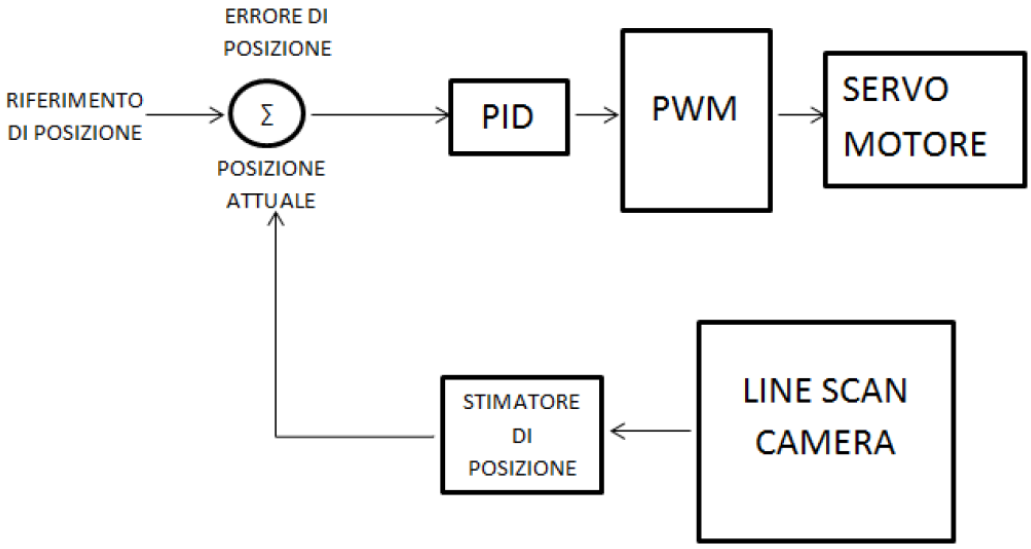


Figura 3.6 – Schema a blocchi del controllo di sterzo

```
#include "MPC5604B.h"
#include "sterzo.h"
#include "define.h"
#include "delay.h"

int8_t LineCenterAct = 0;
uint8_t Flag_PidSterzo = 0;
extern uint16_t steeringCenter;
extern int16_t LineCenterTrue;
extern uint8_t count;
int16_t ErroreSterzo = 0;
int16_t ErrorePreSterzo = 0;
int16_t PSterzo = 0;
int16_t ISterzo = 0;
int16_t DSterzo = 0;
double i_instSterzo = 0;
double KpSterzo = 6.15;
double KiSterzo = 0.0037;
double KdSterzo = 5.5;
uint16_t Steering = 0;
```

Il metodo *uint8_t STERZO (pwmSterzo, steeringCenter)* si occupa di impostare i limiti massimi di movimento dello sterzo per evitarne la rottura causata dal contatto contro lo *chassis* e di posizionarlo esattamente grazie al segnale di PWM proveniente dal canale 4 dell'eMIOS_0.

```
/*Impostazione di un angolo di sterzo tramite comando PWM*/
uint16_t STERZO(uint16_t pwmSterzo, uint16_t steeringCenter)
{
	uint16_t steeringMin = (uint16_t)(steeringCenter - HALF_RANGE_STEER);
	uint16_t steeringMax = (uint16_t)(steeringCenter + HALF_RANGE_STEER);
	/*Impostazione di limiti inferiori e superiori*/
	if (pwmSterzo < steeringMin)
		pwmSterzo = steeringMin;
	if (pwmSterzo > steeringMax)
		pwmSterzo = steeringMax;
	/*Attuazione*/
	EMIOS_0.CH[4].CBDR.R = pwmSterzo;    /*Impostazione dell'angolo di sterzo*/
	return pwmSterzo;
}
```

La funzione *void PidSterzo()* implementa il controllo dello sterzo mediante un controllore PID (proportional, integrative, derivative). La somma pesata dell'azione proporzionale, di quella derivativa e di quella integrale viene utilizzata per regolare la posizione dello sterzo minimizzando l'errore.
La gestione dell'azione integrale è la più delicata: mentre i termini proporzionale e derivativo sono due termini "istantanei" che variano di volta in volta in base all'ingresso e che non tengono di per sé conto della situazione precedente in cui si trovavano, quello integrale è fortemente influenzato dal suo storico e potrebbe arrivare a far saturare l'attuatore. Per desaturare il termine ed evitare gli effetti indesiderati di sovra elongazione si ricorre al così detto ARWU (Anti Reset Wind Up): lo si è implementato imponendo un limite negativo e positivo alla somma pesata dei tre termini, in modo da evitare il caricamento dell'integratore.

```
void PidSterzo (void)
{
	PIT.CH[4].TFLG.R = 0x00000001;
	ErroreSterzo = (int16_t)(64 - LineCenterTrue);
	PSterzo = (int16_t)(ErroreSterzo * KpSterzo);
	if(KiSterzo>0)
	{
		if(((PSterzo + ISterzo + DSterzo)<HALF_RANGE_STEER) || ((PSterzo +
		    ISterzo + DSterzo)>-HALF_RANGE_STEER))  /*Anti Reset Wind Up*/
		{
			i_instSterzo = ErroreSterzo * KiSterzo;
			ISterzo = (int16_t)(ISterzo + i_instSterzo);
		}
	}
	else
		ISterzo = 0;
	if(KdSterzo>0)
	{
		DSterzo = (int16_t)(KdSterzo * (ErroreSterzo - ErrorePreSterzo));
		ErrorePreSterzo = ErroreSterzo;
	}
	else
		DSterzo = 0;
	Steering = (uint16_t)(steeringCenter - (PSterzo + ISterzo + DSterzo));
	STERZO(Steering, steeringCenter);
}
```

Con il metodo *uint16_t STEERING_CALIBRATION(centerStart)* si ha la possibilità di posizionare esattamente lo sterzo in posizione centrale prima di dare lo start al veicolo. Con i pulsanti presenti nella scheda madre infatti si è reso possibile incrementare o decrementare della quantità *DELTA_CALIBRATION_STEERING* la posizione dello sterzo che poi verrà considerata dal programma come sterzo perfettamente dritto.

```
uint16_t STEERING_CALIBRATION(uint16_t centerStart)
{
  uint16_t steeringCenter = centerStart;
  SIU.PCR[68].R = 0x0200;/* Program the drive enable pin of LED1 (PE4) as output*/
  SIU.PCR[69].R = 0x0200;/* Program the drive enable pin of LED2 (PE5) as output*/
  SIU.PCR[70].R = 0x0200;/* Program the drive enable pin of LED3 (PE6) as output*/
  SIU.PCR[71].R = 0x0200;/* Program the drive enable pin of LED4 (PE7) as output*/

  while (1)
  {
      switch(SWITCH())
      {
            case 1:
                  return steeringCenter;
            break;
            case 2:
            break;
            case 3:
                  steeringCenter = (uint16_t)(steeringCenter - DELTA_CALIBRATION_
                  STEERING);         /*Update steering center direction SX*/
                  STERZO(steeringCenter, steeringCenter);
            break;
            case 4:
                  steeringCenter = (uint16_t)(steeringCenter + DELTA_CALIBRATION_
                  STEERING);         /*Update steering center direction DX*/
                  STERZO(steeringCenter, steeringCenter);
            break;
                  default:
            break;
      }
  }
}
```

3.5 MOTORI.C

Il *file* Motori.c contiene tutti i metodi inerenti alla parte di potenza.

```
#include "MPC5604B.h"
#include "motori.h"
#include "define.h"
#include "delay.h"

uint8_t Flag_EncoderSx = 0;
uint8_t Flag_EncoderDx = 0;
int16_t RefSpeedLeft = 3500;
int16_t RefSpeedRight = 3500;
extern double SpeedReadDx;
extern double SpeedReadSx;
int16_t speedLeftAct = 0;
int16_t speedRightAct = 0;
int16_t pwmActLeft = 0;
int16_t pwmActRight = 0;
int16_t erroreLMotori = 0;
int16_t erroreRMotori = 0;
int16_t P_LMotori = 0;
int16_t I_LMotori = 0;
int16_t D_LMotori = 0;
int16_t P_RMotori = 0;
int16_t I_RMotori = 0;
int16_t D_RMotori = 0;
int16_t i_inst_LMotori = 0;
int16_t i_inst_RMotori = 0;
int16_t errore_preLMotori = 0;
int16_t errore_preRMotori = 0;
double KpSpeed = 5.9;
double KiSpeed = 0.01;
double KdSpeed = 0.0;
```

Il metodo *void MOTORS(pwmLeft, pwmRight)* attiva e regola il moto dei motori in corrente continua mediante i canali 2, 3, 6 e 7 del modulo eMIOS_0 e le relative funzioni PWM.

```
void MOTORS(int16_t pwmLeft, int16_t pwmRight)
{
	/*Conversione su base 1000*/
	pwmLeft = (int16_t)(pwmLeft * PWM_MAX_DRIVE / 1000);
	pwmRight = (int16_t)(pwmRight * PWM_MAX_DRIVE / 1000);

	/*Impostazione di limiti inferiori e superiori*/
	if (pwmLeft < -999)
		pwmLeft = -999;
	if (pwmRight < -999)
		pwmRight = -999;
	if (pwmLeft > 999)
		pwmLeft = 999;
	if (pwmRight > 999)
		pwmRight = 999;
	if (pwmLeft > -1)                          /*avanti sinistro*/
	{
		EMIOS_0.CH[2].CBDR.R = 0;              /*IN2 = L*/
		EMIOS_0.CH[6].CBDR.R = pwmLeft;        /*IN1 = H*/
	}
```

```
        if (pwmLeft < 0)                                /*indietro sinistro*/
        {
                EMIOS_0.CH[2].CBDR.R = -pwmLeft;        /*IN2 = H*/
                EMIOS_0.CH[6].CBDR.R = 0;               /*IN1 = L*/
        }
        if (pwmRight > -1)                              /*avanti destro*/
        {
                EMIOS_0.CH[3].CBDR.R = 0;               /*IN2 = L*/
                EMIOS_0.CH[7].CBDR.R = pwmRight;        /*IN1 = H*/
        }
        if (pwmRight < 0)                               /*indietro destro*/
        {
                EMIOS_0.CH[3].CBDR.R = -pwmRight;       /*IN2 = H*/
                EMIOS_0.CH[7].CBDR.R = 0;               /*IN1 = L*/
        }
}
```

Le funzioni *void EncoderSx_Int()* e *void EncoderDx_Int()* sono i metodi richiamati dagli *interrupt* degli encoder. Si è deciso di abilitare dei *flag* (*Flag_EncoderSx* e *Flag_EncoderDx*) grazie agli *interrupt* relativi invece di eseguire tutto il programma di misurazione delle velocità, per limitare il più possibile il tempo di permanenza in questi *interrupt*. In questo modo non si rallenteranno gli altri *interrupt* quali quello dello sterzo e quello dei motori. Infatti questi *flag* serviranno nel *loop* infinito nel metodo *main()* dove si calcoleranno effettivamente le velocità delle ruote e dove si riporterà a zero il valore di tali *flag*.

```
void EncoderSx_Int (void)
{
        EMIOS_1.CH[11].CSR.B.FLAG = 1;
        Flag_EncoderSx = 1;
}
void EncoderDx_Int (void)
{
        EMIOS_1.CH[12].CSR.B.FLAG = 1;
        Flag_EncoderDx = 1;
}
```

La funzione *void PidMotori()* implementa il controllo dello sterzo mediante un controllore PID. I fattori relativi ad ognuna di queste tre azioni (PSterzo, ISterzo, DSterzo) sono stati calcolati in modo sperimentale, come nel caso del PID dello sterzo, e si è imposto un limite negativo e positivo alla somma pesata dei tre termini causata dalla saturazione della parte integrale.

```
void PidMotori (void)
{
        PIT.CH[3].TFLG.R = 0x00000001;                  /*Clear Flag Interrupt*/
        speedLeftAct = (int16_t)SpeedReadSx;
        speedRightAct = (int16_t)SpeedReadDx;
        erroreLMotori = (int16_t)(RefSpeedLeft - speedLeftAct);
        erroreRMotori = (int16_t)(RefSpeedRight - speedRightAct);

        P_LMotori = (int16_t)(erroreLMotori * KpSpeed);
        if(KiSpeed>0)
        {
                if(((P_LMotori + D_LMotori + I_LMotori) < PWM_MAX_DRIVE) ||
                  ((P_LMotori + D_LMotori + I_LMotori) > -PWM_MAX_DRIVE))   /*ARWU*/
                {
                        i_inst_LMotori = (int16_t)(erroreLMotori * KiSpeed);
                        I_LMotori = I_LMotori + i_inst_LMotori;
                }
        }
```

```
        else
                I_LMotori = 0;
        if(KdSpeed>0)
        {
                D_LMotori = (int16_t)(KdSpeed * (erroreLMotori - errore_preLMotori));
                errore_preLMotori = erroreLMotori;
        }
        else
                D_LMotori = 0;
        pwmActLeft = (int16_t)(P_LMotori + D_LMotori + I_LMotori);

        P_RMotori = (int16_t)(erroreRMotori * KpSpeed);
        if(KiSpeed>0)
        {
                if(((P_RMotori + D_RMotori + I_RMotori) < PWM_MAX_DRIVE) ||
                  ((P_RMotori + D_RMotori + I_RMotori) > -PWM_MAX_DRIVE))   /*ARWU*/
                {
                        i_inst_RMotori = (int16_t)(erroreRMotori * KiSpeed);
                        I_RMotori = I_RMotori + i_inst_RMotori;
                }
        }
        else
                I_RMotori = 0;
        if(KdSpeed>0)
        {
                D_RMotori =(int16_t)(KdSpeed * (erroreRMotori - errore_preRMotori));
                errore_preRMotori = erroreRMotori;
        }
        else
                D_RMotori = 0;
        pwmActRight = (int16_t)(P_RMotori + D_RMotori + I_RMotori);

        MOTORS(pwmActRight,pwmActLeft);
}
```

Il metodo *void DIFFERENTIAL (steering, steeringCenter, speedRef, *speedRefLeft, *speedRefRight, increase)* implementa un controllo differenziale sulle ruote posteriori per poter aumentare la velocità di curva mantenendo un'ottima stabilità del veicolo.
Nella percorrenza di una curva la ruota esterna si trova infatti a dover seguire una traiettoria di lunghezza maggiore di quella della ruota interna; in presenza del differenziale si aumenta o diminuisce la velocità di riferimento delle ruote proporzionalmente all'angolo di sterzo impostato. Così facendo, affrontando una curva, si ottiene un aumento della velocità della ruota piu esterna e una diminuzione di quella più interna.
A seconda dell'angolo di sterzo attuato la velocità di riferimento varia linearmente consentendo al controllore PID di modificare il comando PWM da generare ai motori: nel programma "m" e "q" sono rispettivamente il coefficiente angolare e l'intercetta delle rette che legano la curvatura alla velocità. L'incremento/diminuzione di velocità è tale da mantenere costante la velocità media delle due ruote ed è scelto sperimentalmente utilizzando un parametro chiamato *increase*.

```
void DIFFERENTIAL(uint16_t steering, uint16_t steeringCenter, int16_t speedRef,
int16_t* speedRefLeft, int16_t* speedRefRight, float increase)
{
        float m, q;

        m = (float)(2 * increase * speedRef / HALF_RANGE_STEER);
        q = (float)(2 * increase * speedRef * steeringCenter / HALF_RANGE_STEER);
        *speedRefLeft = (int16_t)(m * steering + speedRef - q);
        *speedRefRight = (int16_t)(-m * steering + speedRef + q);
}
```

Con le funzioni *uint16_t SPEED_STRAIGHT_CALIBRATION(straightStart)* e *uint16_t SPEED_TURN_CALIBRATION(turnStart)* si ha la possibilità di aumentare o diminuire manualmente le velocità massime di rettilineo e curva prima di dare lo start al veicolo grazie ai pulsantini presenti sulla scheda madre.

```
int16_t SPEED_STRAIGHT_CALIBRATION(int16_t straightStart)
{
  int16_t speedStraight = straightStart;
  SIU.PCR[68].R = 0x0200;/* Program the drive enable pin of LED1 (PE4) as output*/
  SIU.PCR[69].R = 0x0200;/* Program the drive enable pin of LED2 (PE5) as output*/
  SIU.PCR[70].R = 0x0200;/* Program the drive enable pin of LED3 (PE6) as output*/
  SIU.PCR[71].R = 0x0200;/* Program the drive enable pin of LED4 (PE7) as output*/
  while (1)
  {
       switch(SWITCH())
       {
              case 1:
                    return speedStraight;
              break;
              case 2: break;
              case 3:
                    speedStraight = (int16_t)(speedStraight - DELTA_CALIBRATION_
                                                SPEED_STRAIGHT);
                    break;
              case 4:
                    speedStraight = (int16_t)(speedStraight + DELTA_CALIBRATION_
                                                SPEED_STRAIGHT);
              break;
              default: break;
       }
  }
}

int16_t SPEED_TURN_CALIBRATION(int16_t turnStart)
{
  int16_t speedTurn = turnStart;
  SIU.PCR[68].R = 0x0200;/* Program the drive enable pin of LED1 (PE4) as output*/
  SIU.PCR[69].R = 0x0200;/* Program the drive enable pin of LED2 (PE5) as output*/
  SIU.PCR[70].R = 0x0200;/* Program the drive enable pin of LED3 (PE6) as output*/
  SIU.PCR[71].R = 0x0200;/* Program the drive enable pin of LED4 (PE7) as output*/
  while (1)
  {
       switch(SWITCH())
       {
              case 1:
                    return speedTurn;
              break;
              case 2: break;
              case 3:
                    speedTurn = (int16_t)(speedTurn - DELTA_CALIBRATION_
                                                SPEED_TURN);
              break;
              case 4:
                    speedTurn = (int16_t)(speedTurn + DELTA_CALIBRATION_
                                                SPEED_TURN);
              break;
              default: break;
       }
  }
}
```

3.6 MAIN.C

Tutti i programmi eseguibili in linguaggio C devono definire la funzione *main()*: *main()* è un nome speciale e i programmi iniziano l'esecuzione a partire da questa funzione. Il codice sorgente di un programma puo essere diviso su piu *files*, ma per poter creare un eseguibile, uno ed uno solo di questi deve contenere la definizione della funzione *main()*.

```
#include "MPC5604B.h"
#include "IntcInterrupts.h"
#include "pinout.h"
#include "motori.h"
#include "delay.h"
#include "telecamere.h"
#include "define.h"
#include "sterzo.h"

uint8_t i = 0;
uint8_t l = 0;
uint8_t LineCenter;
uint16_t steeringCenter = 0;
extern double KpSterzo;
extern double KiSterzo;
extern double KdSterzo;
extern uint16_t Steering;
extern uint8_t Flag_PidSterzo;
extern uint16_t cameraLow[N_PIXEL];
extern uint16_t cameraHigh[N_PIXEL];
extern int16_t DevArray[127];
extern uint8_t TrovatoStop;
uint16_t LineCenterArray[LEN_ARRAY_CENTRO] = {64,64};
uint16_t LineCenterTrue = 64;
extern uint8_t Flag_PidMotori;
float PerDifferenziale = 0.17;
extern int16_t RefSpeedLeft;
extern int16_t RefSpeedRight;
int16_t SpeedRef = 3500;
int16_t SpeedStraight = 0;
int16_t SpeedTurn = 0;
uint32_t T2 = 0;
uint32_t T1 = 0;
double T = 0;
double SpeedReadSx = 0;
double SpeedReadPreSx = 0;
double SpeedReadDx = 0;
double SpeedReadPreDx = 0;
extern uint8_t Flag_EncoderSx;
extern uint8_t Flag_EncoderDx;
```

In questo metodo si comanda il registro CGM_SC_DC0…2 (System Clock Divider Configuration Registers) che è un modulo di controllo dei divisori del clock del sistema.

```
void initPeriClkGen(void)
{
    CGM.SC_DC[0].R = 0x80;  /* MPC56xxB: Enable peri set 1 sysclk divided by 1 */
    CGM.SC_DC[1].R = 0x80;  /* MPC56xxB: Enable peri set 2 sysclk divided by 1 */
    CGM.SC_DC[2].R = 0x80;  /* MPC56xxB: Enable peri set 3 sysclk divided by 1*/
}
```

Il SWT_CR (Software Watchdog Timer Control Register) riguarda la configurazione e il controllo di questo modulo che può impedire il blocco del sistema in situazioni come quelle di software intrappolato in un *loop* o qualora un'operazione non riesca a terminare. Quando è abilitato, il SWT richiede la periodica esecuzione di una sequenza di manutenzione.

```
void disableWatchdog(void)
{
    SWT.SR.R = 0x0000c520;              /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A;              /* Clear watchdog enable (WEN) */
}
```

Per la gestione dello sterzo e del blocco motori si è deciso di optare per l'utilizzo di *interrupt* per avere la sicurezza che ogni 20 ms arrivi l'impulso relativo al posizionamento del servo motore e che i motori in corrente continua aggiornino le proprie velocità ripetutamente ad intervalli di tempo stabiliti.
Il PIT (Periodic Interrupt Timer) è un array di timer utilizzabili per generare *interrupt*.
Il registro PITMCR (Module Control Register) controlla se i clock dei timer devono essere abilitati e se i timer devono funzionare in modalità di *debug*.
Come possiamo vedere in figura 3.7 impostando MDIS = 0 si abilita il clock del timer, mentre con FRZ = 1 si blocca il timer in modalità di *debug* consentendo allo sviluppatore del software di fermare il processore per monitorare lo stato attuale del sistema.

Offset: 0x000 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MDIS	FRZ
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Figure 24-57. PIT Module Control Register (PITMCR)

Table 24-25. PITMCR field descriptions

Field	Description
MDIS	Module Disable This is used to disable the module clock. This bit should be enabled before any other setup is done. 0 Clock for PIT timers is enabled 1 Clock for PIT timers is disabled (default)
FRZ	Freeze Allows the timers to be stopped when the device enters debug mode. 0 = Timers continue to run in debug mode. 1 = Timers are stopped in debug mode.

Figura 3.7 – PIT Module Control Register

Il registro LDVAL (Timer Load Value Register) invece permette di impostare il valore iniziale del timer dal quale far partire il conto alla rovescia. Infatti ad ogni colpo di clock il valore impostato verrà decrementato di un'unità fino ad arrivare a zero. Arrivato a fine conteggio verrà generato *l'interrupt* e ripartirà il conto alla rovescia dallo stesso valore impostato in precedenza. È anche possibile modificare il valore di partenza, o disabilitando il timer e quindi interrompendo il ciclo in corso e successivamente attivandolo facendo iniziare subito un nuovo conteggio dal nuovo valore di partenza impostato, o aspettando la fine del conteggio e impostando il nuovo valore al conteggio successivo. Per determinare la frequenza dell'*interrupt* che si vuole generare è necessario usare una semplice proporzione partendo dal fatto che il clock usato dal PIT è a 64 MHz.

Il registro TCTRL (Timer Control Register) contiene i bit di controllo di ogni timer: impostando sia TIE (Timer Interrupt Enable Bit) che TEN (Timer Enable Bit) ad 1 le richieste di *interrupt* vengono abilitate ed i timer attivati.

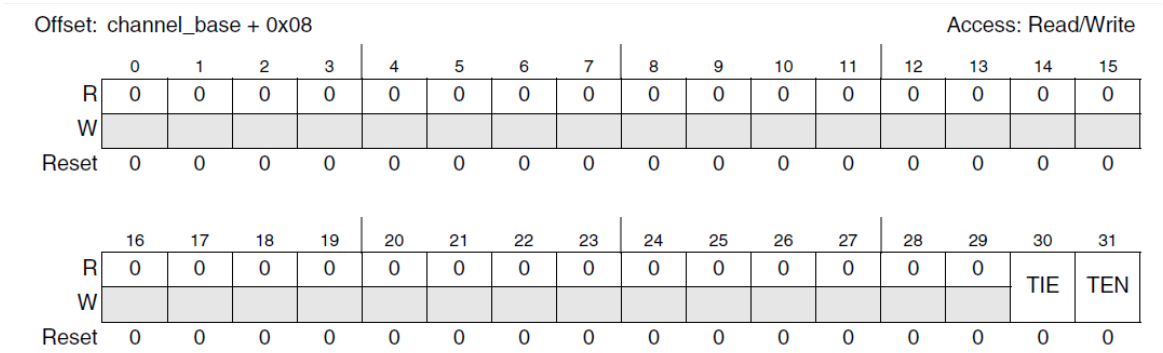


Figure 24-60. Timer Control Register (TCTRL)

Table 24-28. TCTRL field descriptions

Field	Description
TIE	Timer Interrupt Enable Bit 0 Interrupt requests from Timer x are disabled 1 Interrupt will be requested whenever TIF is set When an interrupt is pending (TIF set), enabling the interrupt will immediately cause an interrupt event. To avoid this, the associated TIF flag must be cleared first.
TEN	Timer Enable Bit 0 Timer will be disabled 1 Timer will be active

Figura 3.8 – Timer Control Register

Il registro TFLG (Timer Flag Register) contiene invece i *flag* degli *interrupt*: se la cella 31 (TIF) viene impostata ad 1 si riporta a 0 il *flag* che automaticamente a fine conteggio viene impostato a 1, permettendo così le continue richieste di *interrupt*.

Quindi con questi due metodi si riescono ad inizializzare gli *interrupt* relativi allo sterzo e ai motori.

```
void InitPidSterzo(void)
{
    PIT.PITMCR.R = 0x00000001;  /* 30: MDIS = 0 to enable clock for PITs. */
                                /* 31: FRZ = 1 for Timers stopped in debug mode */
    PIT.CH[4].LDVAL.R = 1280000 - 1;         /*frequenza interrupt 1s=64000000-1*/
    PIT.CH[4].TFLG.R = 0x00000001;           /* clear the TIF flag */
    PIT.CH[4].TCTRL.R = 0x00000003;/* 30: TIE = 1 for interrupt request enabled */
                                   /* 31: TEN = 1 for timer active */
}

void InitPidMotori(void)
{
    PIT.PITMCR.R = 0x00000001;  /* 30: MDIS = 0 to enable clock for PITs. */
                                /* 31: FRZ = 1 for Timers stopped in debug mode */
    PIT.CH[3].LDVAL.R = 440000 - 1;          /*frequenza interrupt 1s=64000000-1/*
    PIT.CH[3].TFLG.R = 0x00000001;           /* clear the TIF flag */
    PIT.CH[3].TCTRL.R = 0x00000003;/* 30: TIE = 1 for interrupt request enabled */
                                   /* 31: TEN = 1 for timer active */
}
```

Si è deciso di utilizzare i canali 3 e 4 dell'array PIT in quanto, dopo aver consultato la tabella *Interrupt Vector Table* nel data sheet del microprocessore, risultavano liberi e disponibili a tale funzionalità.

Le varie richieste di *interrupt* avvengono regolarmente se chiamate una alla volta. Con richieste di *interrupt* multiple, invece, non è possibile soddisfarle tutte contemporaneamente, ma si deve assegnare una precedenza a tali richieste in modo da decidere quale portare a termine prima di tutte. Sono presenti 16 livelli di priorità configurabili per ogni *interrupt*: al livello 0 corrisponde la minima priorità, al livello 15 la massima. Inoltre se la priorità di un *interrupt* in esecuzione è minore di quella di uno appena chiamato, quello in esecuzione viene bloccato per far iniziare quello con precedenza maggiore.
Si è deciso di dare maggiore priorità all'*interrupt* di regolazione dello sterzo per mantenere sempre la giusta direzione del veicolo, poi agli encoder per sapere a quale velocità si sta affrontando il tracciato e successivamente ai motori per accelerare e frenare una volta conosciuti lo sterzo da attuare e la velocità reale.

16.5.2.2 INTC Current Priority Register for Processor (INTC_CPR)

Offset: 0x0008 Access: User read/write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	PRI			
W																																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Figure 16-3. INTC Current Priority Register (INTC_CPR)

Table 16-4. INTC_CPR field descriptions

Field	Description
PRI	Priority PRI is the priority of the currently executing ISR according to the field values defined in Table 16-5.

Table 16-5. PRI values

PRI	Meaning
1111	Priority 15—highest priority
1110	Priority 14
1101	Priority 13
1100	Priority 12
1011	Priority 11
1010	Priority 10
1001	Priority 9
1000	Priority 8
0111	Priority 7
0110	Priority 6
0101	Priority 5
0100	Priority 4
0011	Priority 3
0010	Priority 2
0001	Priority 1
0000	Priority 0—lowest priority

Figura 3.9 – Current Priority Register for Processor

Sono stati creati i metodi *char SWITCH(), void ALL_LED (timeDelay)* e *void STARTING()* per interfacciarsi con la smart car una volta accesa, prima di essere posizionata sul tracciato. Mediante i quattro pulsanti presenti sulla scheda madre si sono riuscite a modificare le principali variabili per il controllo della smart car in modo da cambiare configurazione senza doverla riprogrammare. Inoltre si sono utilizzati i led per avere la conferma di aver modificato tali variabili.
Si ricorda che l'istruzione *break* provoca l'uscita incondizionata da un ciclo, mentre *return* è il meccanismo che consente alla funzione chiamata di restituire un valore al chiamante.

```
char SWITCH(void)
{
   SIU.PCR[64].R = 0x0100;                    /* Program the drive enable */
                                              /* pin of S1 (PE0) as input */
   SIU.PCR[65].R = 0x0100;                    /* Program the drive enable */
                                              /* pin of S2 (PE1) as input */
   SIU.PCR[66].R = 0x0100;                    /* Program the drive enable */
                                              /* pin of S3 (PE2) as input */
   SIU.PCR[67].R = 0x0100;                    /* Program the drive enable */
                                              /* pin of S4 (PE3) as input */
   for(;;)                                    /*Loop until a switch is pressed*/
   {
      if((SIU.PGPDI[2].R & 0x80000000) != 0x80000000)     /*Switch S1*/
      {
            SIU.PGPDI[2].R = 0x00000000;                 /*Reset all switches*/
            return 1;
      }
      if((SIU.PGPDI[2].R & 0x40000000) != 0x40000000)     /*Switch S2*/
      {
            SIU.PGPDI[2].R = 0x00000000;                 /*Reset all switches*/
            return 2;
      }
      if((SIU.PGPDI[2].R & 0x20000000) != 0x20000000)     /*Switch S3*/
      {
            SIU.PGPDI[2].R = 0x00000000;                 /*Reset all switches*/
            return 3;
      }
      if((SIU.PGPDI[2].R & 0x10000000) != 0x10000000)     /*Switch S4*/
      {
            SIU.PGPDI[2].R = 0x00000000;                 /*Reset all switches*/
            return 4;
      }
   }
}

void ALL_LED(uint16_t timeDelay)
{
  SIU.PCR[68].R = 0x0200;                     /* Program the drive enable */
                                              /* pin of LED1 (PE4) as output */
  SIU.PCR[69].R = 0x0200;                     /* Program the drive enable */
                                              /* pin of LED2 (PE5) as output*/
  SIU.PCR[70].R = 0x0200;                     /* Program the drive enable */
                                              /* pin of LED3 (PE6) as output*/
  SIU.PCR[71].R = 0x0200;                     /* Program the drive enable */
                                              /* pin of LED4 (PE7) as output*/
  SIU.PGPDO[2].R &= 0x00000000;               /* Enable LEDs 0000*/
  Delay(timeDelay);
  SIU.PGPDO[2].R |= 0x0f000000;               /* Disable LEDs*/
}

void STARTING(void)
{
      while (1)
      {
            switch(SWITCH())
            {
                  case 1:
                  break;
                  case 2:
                        return;
                  break;
```

```
                    case 3:
                    break;
                    case 4:
                    break;
                          default:
                    break;
              }
        }
}
```

Il metodo principale dell'intero programma è il *void main()*: in esso vengono richiamate tutte le funzioni già generate in un *loop* infinito che inizia con una fase di *starting* e che termina con lo spegnimento del veicolo una volta fermo oltre il segnale di stop.
La funzione comincia con le inizializzazioni dei metodi spiegati in precedenza, poi permette all'utente di modificare, tramite pulsanti sulla scheda madre, velocità di rettilineo, di curva e posizione dello sterzo iniziale. In questa fase di *setup* sono state create delle sequenze luminose grazie ai led presenti nella *mother board* per verificare l'effettiva conferma di aver inserito i valori modificati. Una volta impostati questi dati si dà il via agli *interrupt* e al ciclo *while* infinito contenente il vero e proprio programma di gestione del veicolo.
In questo ciclo infatti si continua ad aggiornare l'array dei centrilinea trovati, si imposta la velocità di rettilineo o di curva (a seconda che la posizione della linea calcolata si trovi entro o al di fuori di un certo *range* centrale dei pixel della telecamera), si attiva il differenziale elettronico se si è in curva e si ferma la smart car se si è oltrepassato il segnale di stop.
Inoltre nel ciclo *while* si sfruttano i *flag* relativi agli *interrupt* degli encoder precedentemente inizializzati. Grazie alla modalità IPM dei canali 11 e 12 del modulo eMIOS_1 è possibile misurare il periodo di tempo che intercorre tra due consecutivi fronti di salita o di discesa di un segnale. Si è deciso di considerare i fronti di salita piuttosto che quelli di discesa, ma è una decisione assolutamente arbitraria. Quando viene rilevato il primo fronte viene salvata la base temporale nei registri A2 e B2; al successivo fronte rilevato questi valori vengono automaticamente spostati nei registri A1 e B1, mentre la nuova base temporale viene risalvata nei registri A2 e B2. La lettura del registro A2 viene fatta da EMIOSA[], mentre quella di B1 da EMIOSB[]. Il *Counter bus* utilizzato è quello generato dal canale 8 dell'eMIOS_1 già inizializzato in precedenza.
Il problema legato alla ripartenza da zero del contatore una volta saturato, viene risolto semplicemente confrontando le due basi temporali, dato che ti tempi di saturazione del contatore sono molto più lunghi rispetto alla successione di due fronti di salita adiacenti del segnale proveniente dall'encoder. Così se il tempo legato al secondo fronte è maggiore di quello del primo fronte il contatore nel frattempo non è saturato, altrimenti sì.

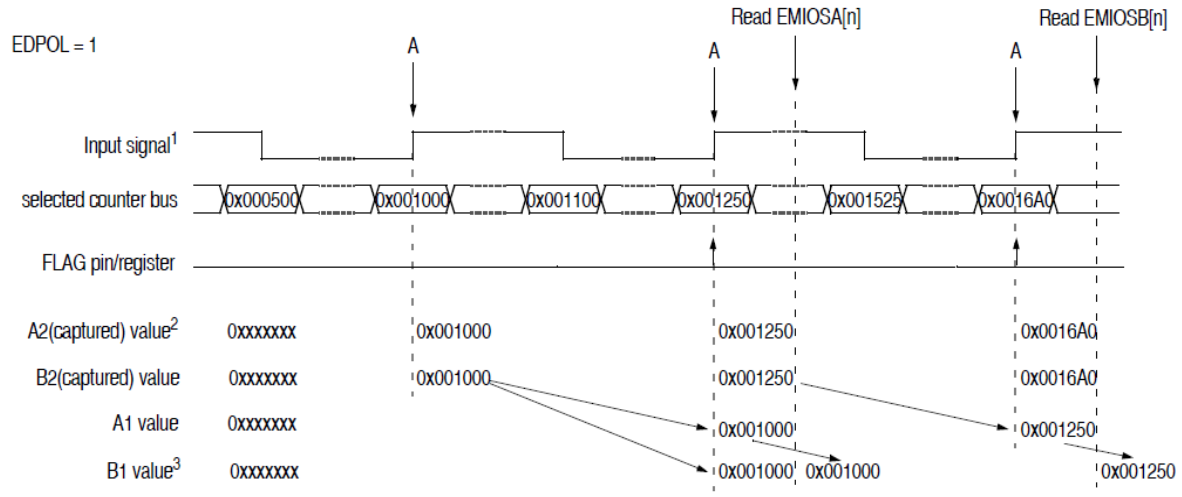


Figura 3.10 – Esempio di funzionamento IPM

```
void main (void)
{
   uint8_t LineCenterPre = 64;

   initModesAndClock();   /* Initialize mode entries and system clock */
   initPeriClkGen();      /* Initialize peripheral clock generation for DSPIs */
   disableWatchdog();     /* Disable watchdog */
   initPads();            /* Initialize pads*/
   initADC();            /* Init. ADC for normal conversions but don't start yet*/
   initEMIOS_0();         /* Initialize eMIOS channels as counter, SAIC, OPWM */
   initEMIOS_1();         /* Initialize eMIOS channels as counter, SAIC, OPWM */
   initEMIOS_0ch0();      /* Initialize eMIOS 0 channel 0 as modulus counter*/
   initEMIOS_1ch8();      /* Initialize eMIOS 1 channel 8 as modulus counter*/
   initEMIOS_0ch23();     /* Initialize eMIOS 0 channel 23 as modulus counter*/
   initEMIOS_0ch2();                  /* Initialize eMIOS 0 channel 0 as OPWM*/
   initEMIOS_0ch3();                  /* Initialize eMIOS 0 channel 0 as OPWM*/
   initEMIOS_0ch4();                  /* Initialize eMIOS 0 channel 0 as OPWM*/
   initEMIOS_0ch6();                  /* Initialize eMIOS 0 channel 0 as OPWM*/
   initEMIOS_0ch7();                  /* Initialize eMIOS 0 channel 1 as OPWM*/
   initEMIOS_1ch11();                 /* Initialize eMIOS 1 channel 11 as IPM,*/
   initEMIOS_1ch12();                 /* Initialize eMIOS 1 channel 12 as IPM*/
   InitPidSterzo();
   InitPidMotori();
   INTC.CPR.R = 0;                    /*INTC Current Priority Register for*/
                                      /*Processor (15 highest - 0 lowest)*/

   for (l=0;l<5;l++)
   {
      ALL_LED(50);
      Delay(50);
   }
   SpeedStraight = SPEED_STRAIGHT_CALIBRATION(SPEED_STRAIGHT);
   for (l=0;l<5;l++)
   {
      ALL_LED(50);
      Delay(50);
   }
   SpeedTurn = SPEED_TURN_CALIBRATION(SPEED_TURN);
   for (l=0;l<5;l++)
   {
      ALL_LED(50);
      Delay(50);
   }
   steeringCenter = STEERING_CALIBRATION(STEERING_CENTER);
   for (l=0;l<10;l++)
   {
      ALL_LED(50);
      Delay(50);
   }
   STARTING();
   ALL_LED(2000);

   RefSpeedLeft = SpeedStraight;
   RefSpeedRight = SpeedStraight;
   SpeedRef = SpeedStraight;

   SIU.PGPDO[2].R |= 0x08000000;      /* Disable LED1*/
   SIU.PGPDO[2].R |= 0x04000000;      /* Disable LED2*/
   SIU.PGPDO[2].R |= 0x02000000;      /* Disable LED3*/
   SIU.PGPDO[2].R |= 0x01000000;      /* Disable LED4*/
   Delay_ms(2000);
```

```
INTC_InstallINTCInterruptHandler(PidSterzo,128,8);     /*Funzione che associa*/
INTC_InstallINTCInterruptHandler(PidMotori,127,5);     /*il metodo da*/
INTC_InstallINTCInterruptHandler(EncoderSx_Int,162,6); /*eseguire al flag di*/
INTC_InstallINTCInterruptHandler(EncoderDx_Int,163,7); /*interrupt generato*/

while(1)
{
    CAMERA(cameraLow);
    LineCenter = FIND_CENTER_LINE(cameraLow,LineCenterPre);
    if((LineCenter!=0)&&(LineCenter!=128))
    {
          for (i=LEN_ARRAY_CENTRO-1;i>0;i--)
                LineCenterArray[i] = LineCenterArray[i-1];
          LineCenterArray[0] = LineCenter;
          LineCenterTrue = 0;
          for(i=0;i<LEN_ARRAY_CENTRO;i++)
                LineCenterTrue = LineCenterTrue + LineCenterArray[i];
          LineCenterTrue = LineCenterTrue/LEN_ARRAY_CENTRO;
          LineCenterPre = LineCenter;
    }
    else
    {
          for (i=LEN_ARRAY_CENTRO-1;i>0;i--)
                LineCenterArray[i] = LineCenterArray[i-1];
          LineCenterArray[0] = LineCenterPre;
          LineCenterTrue = 0;
          for(i=0;i<LEN_ARRAY_CENTRO;i++)
                LineCenterTrue = LineCenterTrue + LineCenterArray[i];
          LineCenterTrue = LineCenterTrue/LEN_ARRAY_CENTRO;
    }
    if(!TrovatoStop)
    {
          if((LineCenterTrue>55) && (LineCenterTrue<73))
                SpeedRef = SpeedStraight;
          else
                SpeedRef = SpeedTurn;
          if((Steering>steeringCenter+100) || (Steering<steeringCenter-100))
                DIFFERENTIAL(Steering, steeringCenter, SpeedRef, &RefSpeedLeft,
                             RefSpeedRight, PerDifferenziale);
          else
          {
                RefSpeedLeft = SpeedRef;
                RefSpeedRight = SpeedRef;
          }
    }
    else
    {
          SpeedRef  = SpeedRef - 100;
          if(SpeedRef<500)
          {
                PIT.CH[3].TCTRL.B.TIE = 0; /* Interrupt requests disabled*/
                MOTORS(0,0);
          }
    }
    if(Flag_EncoderSx==1)
    {
          T2 = EMIOS_1.CH[11].CADR.R;     /* Read Captured input*/
          T1 = EMIOS_1.CH[11].CBDR.R;     /* Clear flag */
          if(T2>T1)
          {
                T = (T2 - T1)*1.0;
```

```
                T = ((T * 0.06)/60000);   /*Tempo in secondi */
                SpeedReadSx = (14.5/T);   /*14.5 = spazio tra le lamelle */
            }
            else if(T1>T2)
            {
                T = (60000 - (T1 - T2))*1.0;
                T = ((T * 0.06)/60000);   /* Tempo in secondi*/
                SpeedReadSx = (14.5/T);   /* 14.5 = spazio tra le lamelle */
            }
            else
                SpeedReadSx = SpeedReadPreSx;
            SpeedReadPreSx = SpeedReadSx;
            Flag_EncoderSx=0;
        }
        if(Flag_EncoderDx==1)
        {
            T2 = EMIOS_1.CH[12].CADR.R;     /* Read Captured input */
            T1 = EMIOS_1.CH[12].CBDR.R;     /* Clear flag */
            if(T2>T1)
            {
                T = (T2 - T1)*1.0;
                T = ((T * 0.06)/60000);   /* Tempo in secondi */
                SpeedReadDx = (14.5/T);   /* 14.5 = spazio tra le lamelle */
            }
            else if(T1>T2)
            {
                T = (60000 - (T1 - T2))*1.0;
                T = ((T * 0.06)/60000);   /* Tempo in secondi */
                SpeedReadDx = (14.5/T);   /* 14.5 = spazio tra le lamelle */
            }
            else
                SpeedReadDx = SpeedReadPreDx;
            SpeedReadPreDx = SpeedReadDx;
            Flag_EncoderDx=0;
        }
    }
}
```

Conclusioni

Lo scopo ultimo di questo elaborato è quello di fornire una visione approfondita sulla gestione e l'ottimizzazione del software in quanto, a parità di hardware fornito, è proprio il programma che decide le sorti della gara.
Il documento risulta più completo rispetto ai semplici manuali tecnici, ed ha come obiettivi quello di far prendere consapevolezza di come funzionano i componenti in esame, rimandando poi alla letteratura tecnica per chiarimenti più dettagliati, ma soprattutto di far comprendere al lettore il perché nel software si utilizzino certi costrutti.
Ci si augura che la trattazione abbia soddisfatto le aspettative del lettore e che possa essere stata d'aiuto per risolvere dubbi soprattutto a studenti che si rivolgono a questa sfida, magari per la prima volta.
Gli argomenti trattati possono servire come base di partenza per i futuri concorrenti, ciò non toglie loro la possibilità di cimentarsi in soluzioni alternative come l'utilizzo di un accelerometro, la realizzazione di un algoritmo per riconoscere le curve dalle *chicane* e il miglioramento dei dispositivi introdotti e dell'algoritmo utilizzato.
Questa per me è stata sicuramente un'occasione per collegare e mettere sul campo tutte le conoscenze base della meccatronica ed è stata anche un'esperienza utile per attuare un modo diverso di approciarsi ai problemi incontrati, più pratico e meno teorico.
Per questo motivo ritengo che tali esperienze siano formative e permettano a noi aspiranti ingegneri di capire molti aspetti del lavoro che andremo a svolgere, scoprendo modalità corrette di approccio ai problemi dei sistemi meccatronici.

Bibliografia e Sitografia

1. Freescale Semiconductor, MPC5604B/C Microcontroller Reference Manual, Rev. 8.1, May 2012
2. Taos, TSL1401CL 128x1 linear sensor array with hold, July 2011
3. Freescale Semiconductor, MC33931 Data Sheet, Rev. 3.0, June 2012
4. M. Zigliotto, Dispense del corso di Fondamenti di Macchine ed Azionamenti Elettrici, 2013
5. A. Pretto, Dispense del corso di Linguaggi di Programmazione per Sistemi Industriali, 2012
6. https://community.freescale.com
7. Futaba (http://www.futabarc.com)
8. Standard Motors (http://www.standardmotor.net/)
9. http://en.wikipedia.org/wiki/Freescale_Semiconductor